Contents

PmagPy Cookbook

January 14, 2014

Dear Reader,

This documentation is updated from that in the book entitled *Essentials of Paleomagnetism* by Tauxe et al., (2010). This cookbook was designed as a companion website to the the book Essentials of Paleomagnetism, 2nd Web Edition. Chapter references to this companion book are, for example, "Essentials Chapter X".

There are many chefs who contributed to this work, in particular, the MagIC Database Team (Cathy Constable, Anthony Koppers, Rupert Minnett, Nick Jarboe, and Ron Shaar).

Lisa Tauxe
Scripps Institution of Oceanography
La Jolla, CA 92093
January 14, 2014
`http://magician.ucsd.edu/~ltauxe/`

Chapter 1

MagIC Quick Start

To skip all the details and just start using MagIC.py to upload data into the MagIC database, you follow these steps:

1. Download and install the Enthought Canopy distribution of Python `https://www.enthought.com/products/canopy/`. Canopy Express (the free version) contains everything you need, but if you are at a degree-granting educational institution you can request an academic license that gives you access to additional features and bundled packages.

2. Download the PmagPy software package: unzip the zip file, extract all the files to a file on your desktop (or wherever) and double click on the *install_MacOS* or *install_Windows* file depending on your operating system. (If you are a Linux user, just copy the files to some directory and put it in your path - you know what to do.)

3. For example data files, download the Datafiles_2.0 directory and unzip it.

4. Create a MagIC Project Directory.

5. Find your command line and type **MagIC.py**. [Windows users must have already created the MagIC directory before starting **MagIC.py**.]

6. Import your data files and combine your measurements.

7. Use one of the plotting and analysis tools (demagnetization data, Thellier-type experiments, etc.).

8. Assemble your results into file for importing into a MagIC smartbook and thence to the MagIC database.

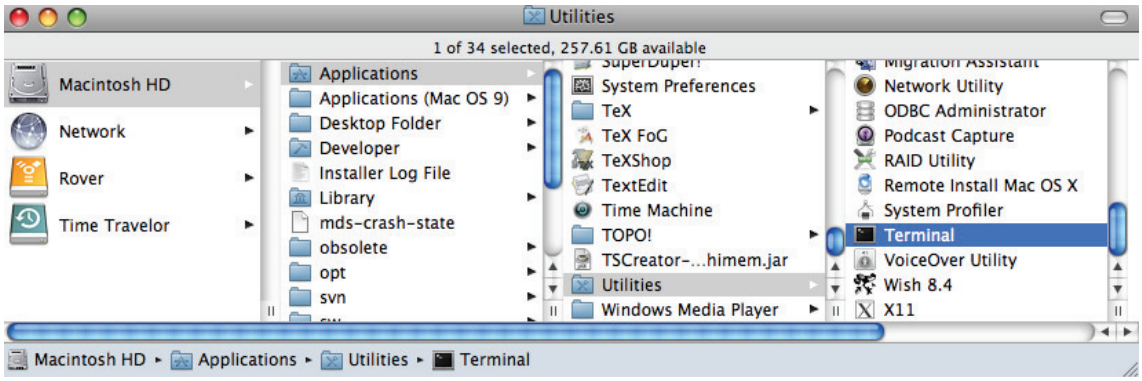
Chapter 2

Survival computer skills

The 'Py' part of 'PmagPy' stands for Python, the language in which all the code is written. It is not essential, but it is helpful to understand a bit about computer operating systems and the Python language when using PmagPy because no one should be using programs as black boxes without understanding what they are doing. As all the programs are open source, you have the opportunity to look into them. If you understand a bit about how computers work yourself, you will be able to follow along what they are doing and even modify them to work better for you. So in this chapter you will find a brief introduction to the computer skills necessary for using the programs properly. I have tried very hard to make this tutorial operating system independent. All the examples should work equally well on Mac OS, Windows and Unix-type operating systems. For an introduction to programming in Python, see Chapter 6. For now, we're just interested in getting started with PmagPy.

2.1 Finding your command line

If you are not using a Unix-like computer (*NIX), you may never have encountered a command line. While the MagIC.py Graphical User Interface (GUI) is an attempt to make life as easy for you by constructing commands for you, you still need to find the command line to start it up. Under the MacOS X operating system look for the Terminal application in the Utilities folder within the Applications folder:



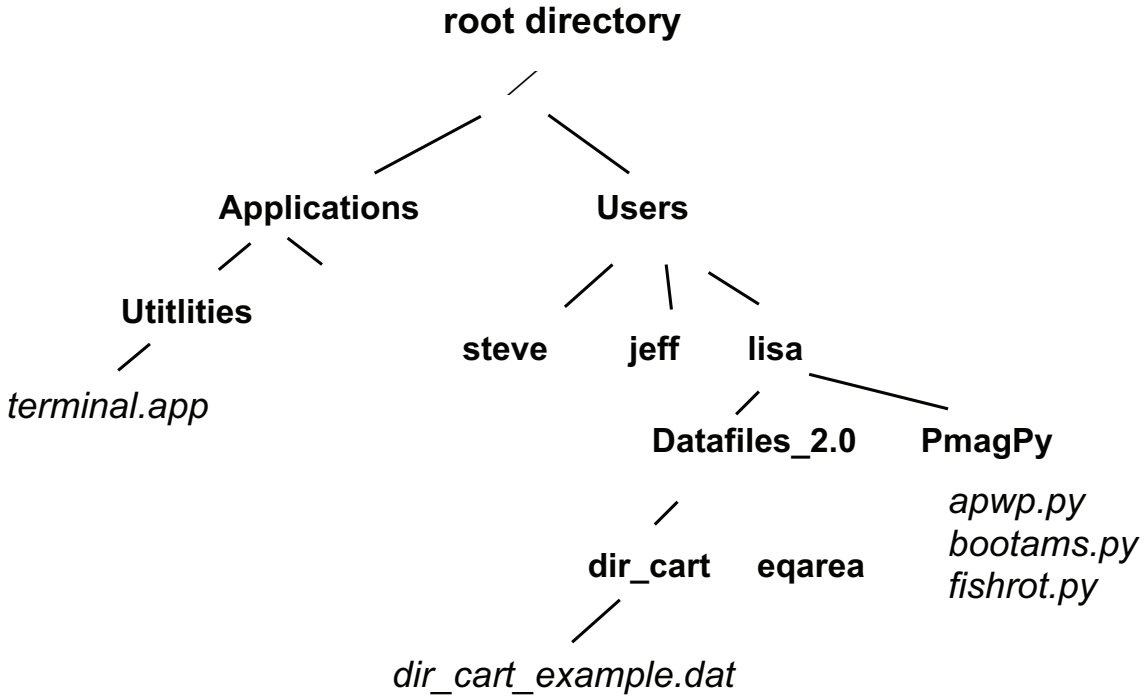
Under the Windows operating system, find the program called: Command Prompt in the Start menu:

Note that the location of this program varies on different computers, so you may have to hunt around a little to find yours. Also, the actual "prompt" will vary for different machines. The MacOS (C-shell) and Windows command line windows look like these:

The Unix (and Mac OS) Bash shell has a $ sign as a prompt. I will use the % prompt in the examples here, but any of these command line prompts will work.

2.2 File systems

When you open one of these terminal windows, you are in your "home" directory. Fundamental to all operating systems is the concept of directories and files. On windows-based operating systems (MacOS or Windows), directories are depicted as "folders" and moving about is accomplished by clicking on the different icons. In the world of terminal windows, the directories have names and are arranged in a hierarchical sequence with the top directory being the "root" directory, known as "/" (or C:'backslash' in Windows) and the file system looks something like this:



Within the root directory, there are subdirectories (e.g. **Applications** and **Users** in bold face). In any directory, there can also be "files" (e.g. *dir_cart_example.dat* in italics). To refer to directories, the operating system relies on what is called a "pathname". Every object has an "absolute" pathname which is valid from anywhere on the computer. The absolute pathname in *NIX always begins from the root directory / and in DOS

(the operating system working in the Windows command line window), it is C:'backslash'.

The absolute pathname to the home directory **lisa** in the figure is **/Users/lisa**. Similarly, the absolute pathname to the directory containing **PmagPy** scripts would be **/Users/ltauxe/PmagPy**. There is also a "relative" pathname, which is in reference to the current directory (the one you are 'sitting' in). If user "lisa" is sitting in her home directory, the relative pathname for the file *dir_cart_example.dat* in the directory **Datafiles_2.0** would be *Datafiles_2.0/dir_cart/dir_cart_example.dat*. When using relative pathnames, it is useful to remember that **./** refers to the current directory and **../** refers to the directory "above". Also, lisa's home directory would be ∼lisa, or if you are logged in as lisa yourself, then it is just ∼.

2.3 Moving around in the file system

Now that you have found your command line and are comfortable in your home directory, you can view the contents of your directory with the Unix command **ls** or the DOS command **dir**. You can make a new directory with the command

```
%mkdir NEW_DIRECTORY_NAME
```

This works in both Unix and DOS environments) and you can move into your new directory with the command

```
% cd NEW_DIRECTORY_NAME
```

To move back up into the home directory, just type **cd ..** remembering that .. refers to the directory above. Also, **cd** by itself will transport you home from where ever you are (there's no place like home....). You can also change to any arbitrary directory by specifying the full path of the destination directory.

2.4 Wildcards

Unix and DOS have the ability to refer to a number of files and/or directories using "wildcards". The wildcard for a single character is "?" and for any number of characters is "*". For example, to refer to all the files with ".py" in their name in the PmagPy directory in my home directory, I would type:

```
% ls PmagPy/*.py
apwp.py bootams.py fishrot.py.....
```

2.5 Redirecting input and output

Programs that operate at the command line level print output to the screen and read input from the keyboard. This is known as "standard input and output" or "standard I/O". One of the nicest things about working at the command line level is the ability to redirect input and output. For example, instead of typing input to a program with the keyboard, it can be read from a file using the symbol <. Output can either be printed to the screen (standard output), redirected into a file using the symbol >, appended to the end of a file with >> or used as input to another program with the pipe operator (|).

2.6 Text editors

Text editing is a blessing and a curse. You either love it or hate it and in the beginning, and if you are used to programs like Word, you will certainly hate it. (And if you are used to a decent text editor, you will hate Word!). There are many ways of editing text and the subject is beyond the scope of this documentation. But you can't use Word because the output is in a weird format that no scripting languages read easily. So you have to use an editor that will produce a plain (ascii) file, like Notepad or TextWrangler. The latter is freeware available for Macs and the former comes standard in the Windows operating system.

2.7 Installing Python and PmagPy

Python can be painful to install (but so can all other programming environments). There are a few recipes that work for Mac OS (at least for 10.4 and later) and for Windows. These recipes and ingredients are available through the website:

`http://earthref.org/PmagPy`.

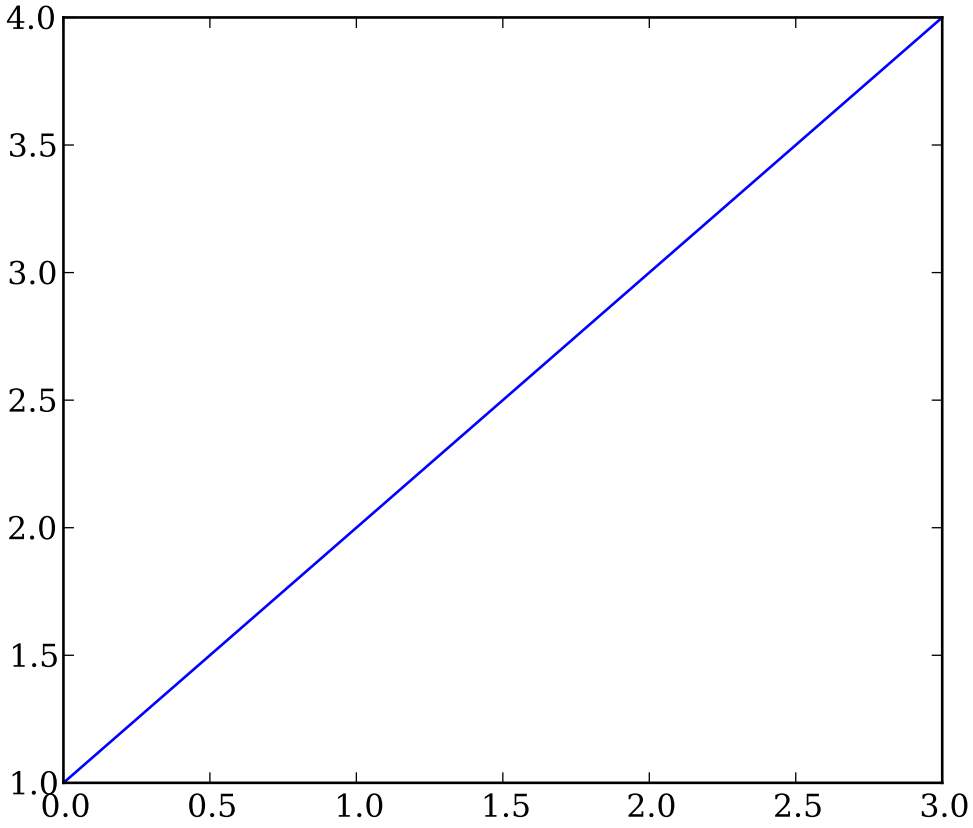
Once Python is installed, find your command line with its prompt. After the command prompt, type: **python** to start the interactive python shell. To get back out again, hold down the control key while typing the letter "d". On Windows machines, you may have to substitute a letter "c" to achieve the same effect. This action will be referred to with 'ctrl-D' here.

Everyone should now have the >>> Python prompt. Here is a transcript of a python interpreter session which should make a simple graph:

```
% python
```

```
Enthought Python Distribution -- www.enthought.com
Version: 7.1-2 (32-bit)

Python 2.7.2 |EPD 7.1-2 (32-bit)| (default, Jul 27 2011, 13:29:32)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "packages", "demo" or "enthought" for more information.
>>> import matplotlib
>>> matplotlib.use("TkAgg")
>>> import pylab
>>> pylab.plot([1,2,3,4])
[<matplotlib.lines.Line2D object at 0x64eb270>]
>>> pylab.show()
```

which produces the fascinating graph:



To kill the interpreter, close the plot window (red button) and type ctrl-D.

If you don't get something about Enthought Python, you are probably using the standard Mac Os version, which has none of the whistles and bells we want (plotting, numerical packages, etc.), and you'll need to set your path properly. Look for hints at: `http://earthref.org/PmagPy`. For more on how to program in Python, see Chapter 6.

Chapter 3

The MagIC database and file formats

A number of the programs in **PmagPy** were developed to take advantage of the MagIC database and aid getting data in and out of it. So, we need some basic understanding of what MagIC is and how it is structured. MagIC is an Oracle database that is part of the EarthRef.org collection of databases and digital reference material. Anyone interested in the MagIC database should first become a registered EarthRef.org user. To do this, go to `http://earthref.org` and click on the **Register** link in the **Topmenu**. Registration is not required for access to data or browsing around, but is required for uploading of data into the MagIC database, something which we sincerely hope you will have a chance to do. By registering, you will also be kept informed of our progress through bi-monthly newsletters and other email alerts. After you register, go to `http://earthref.org/MAGIC`.

3.1 Perusing the existing data

The MagIC database is designed to have two web portals, one for paleomagnetic data (PMAG) and one for rock magnetic data (RMAG). If you click on the PMAG PORTAL link, you will see a link labeled "Open the data tab to Start searching" which, if you click on it, takes you to the main MagIC search engine. You can look for data from a particular reference by typing in the author's name in the "Reference Text Search" box, or search by any column with a little magnifying glass icon. Once the data you want have been identified, you can download the whole data set for example by clicking on the icon for the textfile:

After downloading, the data can be unpacked and examined using various tools in the **PmagPy** programs described later.

3.2 Uploading data to the database

Paleomagnetic and rock magnetic data are collected and analyzed in a wide variety of ways with different objectives. Data sets can be extremely large or can be the barest boned data summaries published in legacy data tables. The goal of MagIC has been to have the flexibility to allow a whole range of data including legacy data from publications or other databases to new studies which include all the measurements, field photos, methodology, and so on. The general procedure for the future will be to archive the data at the same time that they are published. So, to smooth the path, it is advisable to put your data into the MagIC format as early in the process as possible. All data that enters the database must pass through an Excel spreadsheet, called the MagIC Console. The Console program allows you to open a "smartbook" comprised of some 30 tables each with dozens of column headings (meta-data). Data are assembled in a MagIC template file, checked for consistency and completeness, then exported to text and excel files. These can then be uploaded into the MagIC database. Data can either be entered directly into the Console, or can be prepared into a strictly formatted ascii file behind the scenes which can be imported into the Console late in the process. Here, we illustrate the use of a number of programs (the PmagPy package in Python) to facilitate the 'behind the scenes' process.

3.3 Structure of the database tables

The MagIC database is organized around a series of data tables. The complete data model can be found here: `http://earthref.org/MAGIC/metadata.htm`.

Each MagIC table has a one line header of the form:

tab **table_name**

"tab" (or "tab delimited") means that the table is tab delimited. In theory other delimiters are possible, but **PmagPy** only uses tab delimited formats. The **table_name** is one of the table names. The tables are of four general types: EarthRef tables (**er_**) shared in common with other EarthRef databases, MagIC tables (**magic_**) common to both rock magnetic and

paleomagnetic studies, Paleomagnetic tables (**pmag_**), data reduction useful in paleomagnetic studies, Rock magnetic tables (**rmag_**), data reduction useful for rock magnetic studies. Most studies use only some of these tables. Here are some useful tables for a typical paleomagnetic study (starred are required in all cases):

table	Brief description
er_locations*	geographic information about the location(s) of the study
er_sites	locations, lithologic information, etc. for the sampling sites
er_samples	orientation, sampling methods, etc. for samples
er_specimens	specimen weights, volumes
er_ages	age information.
er_images	images associated with the study (field shots, sample photos, photomicrographs, SEM images, etc.
er_citations*	citation information
er_mailinglist	contact information for people involved in the study
magic_measurements	measurement data used in the study
magic_methods*	methods used in the study
magic_instruments	instruments used in the study
pmag_specimens	interpretations of best-fit lines, planes, paleointensity, etc.
pmag_samples	sample averages of specimen data
pmag_sites	site averages of sample data
pmag_results	averages, VGP/V[A]DM calculations, stability tests, etc.
pmag_criteria	criteria used in study for data selection
rmag_susceptibility	experiment for susceptibility parameters
rmag_anisotropy	summary of anisotropy parameters
rmag_hysteresis	summary of hysteresis parameters
rmag_remanence	summary of remanence parameters
rmag_results	summary results and highly derived data products (critical temperatures, etc.)
rmag_criteria	criteria used in study for data selection

The second line of every file contains the column headers (meta-data) describing the included data. For example, an **er_sites** table might look like this:

tab **er_sites**					
er_site_name	er_location_name	site_lithology	site_type	site_lat	site_lon
AZ01	Azores	basalt	lava flow	37.80	-25.80
...					

Although data can be entered directly into the MagIC Console, it is easier to generate the necessary tables as a by-product of ordinary data

processing without having to know details of the meta-data and method codes. The following section describes how to use the **PmagPy** software for data analysis and generate the MagIC data tables automatically for the most common paleomagnetic studies involving directions and/or paleointensities.

3.4 A word about method codes

The MagIC database tags records with "method codes" which are short codes that describe various methods associated with a particular data record. The complete list is available here: `http://earthref.org/MAGIC/methods.htm`. Most of the time, you do not need to know what these are (there are over a hundred!), but it is helpful to know something about them. These are divided into several general categories like 'geochronology methods' and 'field sampling methods'. Method codes start with a few letters which designate the category (e.g., GM or FS for geochronogy and field sampling respectively). Then there is a second part and possibly also a third part to describe methods with lesser or greater detail. The current (version 2.4) method codes that describe various lab treatment methods to give you a flavor for how they work are listed in this table:

LT-AF-D	Lab Treatment	Alternating field: Double demagnetization with AF along X,Y,Z measurement followed by AF along -X,-Y,-Z measurement
LT-AF-G	Lab Treatment	Alternating field: Triple demagnetization with AF along Y,Z,X measurement followed by AF along Y and AF along Z measurement
LT-AF-I	Lab Treatment	Alternating field: In laboratory field
LT-AF-Z	Lab Treatment	Alternating field: In zero field
LT-CHEM	Lab Treatment	Cleaning of porous rocks by chemical leaching with H
LT-FC	Lab Treatment	Specimen cooled with laboratory field on
LT-HT-I	Lab Treatment	High temperature treatment: In laboratory field
LT-HT-Z	Lab Treatment	High temperature treatment: In zero field
LT-IRM	Lab Treatment	IRM imparted to specimen prior to measurement
LT-LT-I	Lab Treatment	Low temperature treatment: In laboratory field
LT-LT-Z	Lab Treatment	Low temperature treatment: In zero field
LT-M-I	Lab Treatment	Using microwave radiation: In laboratory field
LT-M-Z	Lab Treatment	Using microwave radiation: In zero field
LT-NO	Lab Treatment	No treatments applied before measurement
LT-NRM-APAR	Lab Treatment	Specimen heating and cooling: Laboratory field anti-parallel to the NRM vector
LT-NRM-PAR	Lab Treatment	Specimen heating and cooling: Laboratory field parallel to the NRM vector
LT-NRM-PERP	Lab Treatment	Specimen heating and cooling: Laboratory field perpendicular to the NRM vector
LT-PTRM-I	Lab Treatment	pTRM tail check: After zero field step, perform an in field cooling
LT-PTRM-MD	Lab Treatment	pTRM tail check: After in laboratory field step, perform a zero field cooling at same temperature
LT-PTRM-Z	Lab Treatment	pTRM tail check: After in laboratory field step, perform a zero field cooling at a lower temperature
LT-T-I	Lab Treatment	Specimen cooling: In laboratory field
LT-T-Z	Lab Treatment	Specimen cooling: In zero field
LT-VD	Lab Treatment	Viscous demagnetization by applying MU-metal scree
LP-X	Lab Treatment	Susceptibility
LT-ZF-C	Lab Treatment	Zero field cooled, low temperature IRM imparted
LT-ZF-CI	Lab Treatment	Zero field cooled, induced M measured on warming

Chapter 4

MagIC.py

[MagIC]

MagIC.py is an umbrella program that links many MagIC related functions of the PmagPy Software Package in a user-friendly graphical user interface (GUI). While it is invoked with a command line call, this can be done from any directory (with no spaces in the path). It generates calls to the programs described in Chapter 5 so you don't have to. It allows importing of many lab and instrument formats, plotting of a variety of data and doing the data processing and book-keeping required to create files ready for uploading into the MagIC Console software.

4.1 Preparing for MagIC

4.1.1 Project Directory

When you invoke **MagIC.py**, the first step is to choose a 'Project Directory'. For each study, create a directory with a name that relates to that study. Here I will call it *ThisProject*. This is where you will collect and process all the rock and paleomagnetic data for a given study, usually a publication. The project directory name should have NO SPACES and be placed on the hard drive in a place that has NO spaces in the path. Under certain Windows versions, this means you should not use your home directory, but create a directory called for example: D:\MyPmagProjects and put *ThisProject* there.

Inside the *ThisProject* directory, create two additional directories: *MyFiles* and *MagIC*. All the files that you want to import into the MagIC format should be placed in *MyFiles* and you should just leave *MagIC* alone

unless you really know what you are doing. The *Project Directory* that **MagIC.py** seeks is that *MagIC* directory.

Your Directory tree might look like this now:

4.1.2 What goes in *MyFiles*

Field and sampling information

There is an astounding number of different ways that paleomagnetists document data in the field and in the lab. This variety is met with a large number of method codes that describe sampling and orientation procedures (see `http://earthref.org/MAGIC/methods.htm` for a complete description). The MagIC database expects sample orientations to be the azimuth and plunge of the fiducial arrow used for measurement (see [Essentials, Chapter 9]) and the orientation of the bedding to be dip direction and downward dip so no matter what your own preference is, it must be translated into the standard MagIC convention for use with the PmagPy programs and with the MagIC.py GUI.

To make the conversion from notebook information to the MagIC format, you can create a tab delimited file (orient.txt format). This file should have all the information for a single location *sensu MagIC*. [A location is a stratigraphic section, a sampling region, an drill core, and so on.] MagIC doesn't really care what your location name is, but use the same location name every time you are asked for it, because it really ties your dataset together. The first line of the *orient.txt* file should be a header with the word 'tab' in the first column and the desired location name in the second column:

orient.txt

	1	2	3	4	5	6	7	8
1	tab	North Shore Volcanics						
2	sample_name	mag_azimuth	field_dip	sample_class	sample_type	sample_lithology	lat	long
3	ns034a	354	18	igneous:extrusive	lava flow	basalt	47.01668	-91.66029
4	ns034b	4	31	igneous:extrusive	lava flow			
5	ns034c	15	26	igneous:extrusive	lava flow			
6	ns034d	156	53	igneous:extrusive	lava flow			
7	ns034e	272	32	igneous:extrusive	lava flow			
8	ns034f	343	62	igneous:extrusive	lava flow			
9	ns034g	299	49	igneous:extrusive	lava flow			
10								

The next row has the names of the columns. The required columns are: sample_name, mag_azimuth, field_dip, date, lat, long, sample_lithology, sample_type, sample_class) but there are a number of other possible columns (e.g., Optional Fields in orient.txt formatted files are: [date, shadow_angle, hhmm], date, stratigraphic_height, [bedding_dip_direction, bedding_dip], [image_name, image_look, image_photographer], participants, method_codes, site_name, and site_description, GPS_Az]). Column names in brackets must be supplied together and the data for stratigraphic_height are in meters. Also note that if these are unoriented samples, just set mag_azimuth and field_dip to 0.

If there is a simple and consistent relationship between the site name and

the sample name (e.g., sample ns034a belongs to site ns034), you do not need to specify a site name here as it will be parsed by orientation_magic.py when it gets imported to the MagIC format. However, many investigators have no such consistent naming scheme. Moreover, in some cases, groups of samples or initial site designations need to be re-grouped for averaging. For example, if it becomes clear that a sequence of lava flows were erupted over a short period of time and should be averaged together, you would need a new site name for all the samples. Note that there are different understandings of the term **site** in the paleomagnetic community. We adhere to the MagIC definition of "site", which is:

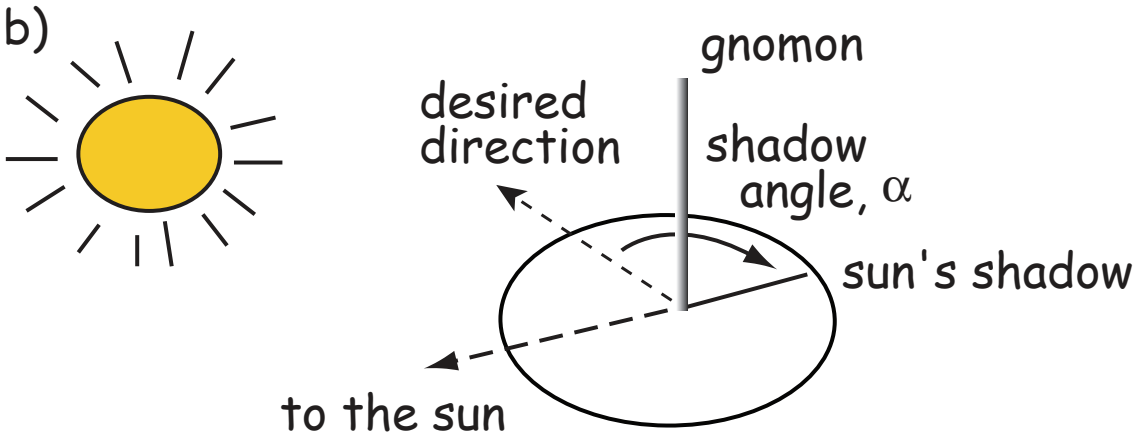
> site: a group of samples that are homogeneous with respect to the property being measured.

When a site has no simple relationship to the sample names, a column named site_name in the *orient.txt* file can be used with the site name filled in for every sample.

It is handy to document the lithology, type and material classification information required by MagIC. These are all controlled vocabularies listed at `http://earthref.org/MAGIC/shortlists.htm`. For archaeological materials, set the lithology to "Not Specified".

Put in stratigraphic height, sun compass, differential GPS orientation information under the appropriate column headings. You can also flag a particular sample orientation as suspect, by having a column 'sample_flag' and setting it to either 'g' for good or 'b' for bad. Other options include documenting digital field photograph names and who was involved with the sampling.

For Sun Compass measurements, supply the shadow_angle, date and time. The date must be in mm/dd/yy format. If you enter the time in local time, be sure you know the offset to Universal Time as you will have to supply that when you import the file. Also, only put data from one time zone in a single file. The shadow angle should follow the convention shown in this figure (from Tauxe et al., 2010):



Supported sample orientation schemes:

There are options for different orientation conventions (drill direction with the Pomeroy orientation device [drill azimuth and hade] is the default), different naming conventions and a choice of whether to automatically calculate the IGRF value for magnetic declination correction, supply your own or ignore the correction. The program generates *er_samples.txt* and er_sites.txt files. Be warned that existing files with these names will be overwritten.

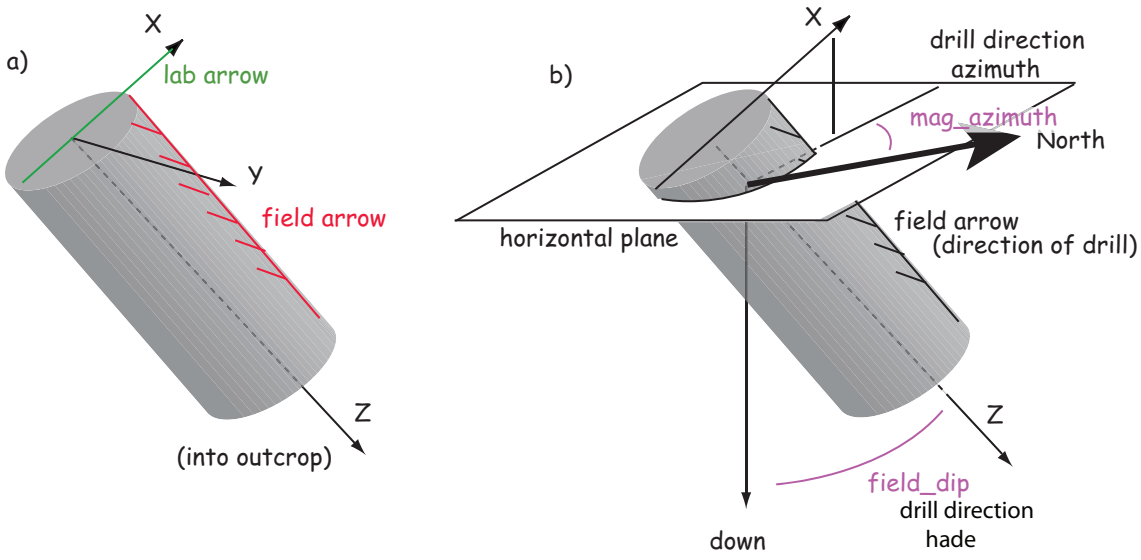
All images, for example outcrop photos are supplied as a separate zip file. image_name is the name of the picture you will import, image_look is the "look direction" and image_photographer is the person who took the picture. This information will be put in a file named er_images.txt and will ultimately be read into the er_image table in the console where additional information must be entered (keywords, etc.).

Often, paleomagnetists note when a sample orientation is suspect in the field. To indicate that a particular sample may have an uncertainty in its orientation that is greater than about 5°, enter SO-GT5 in the method_codes column and any other special codes pertaining to a particular sample from the method codes table. Other general method codes can be entered later. Note that unlike date and sample_class, the method codes entered in orient.txt pertain only to the sample on the same line.

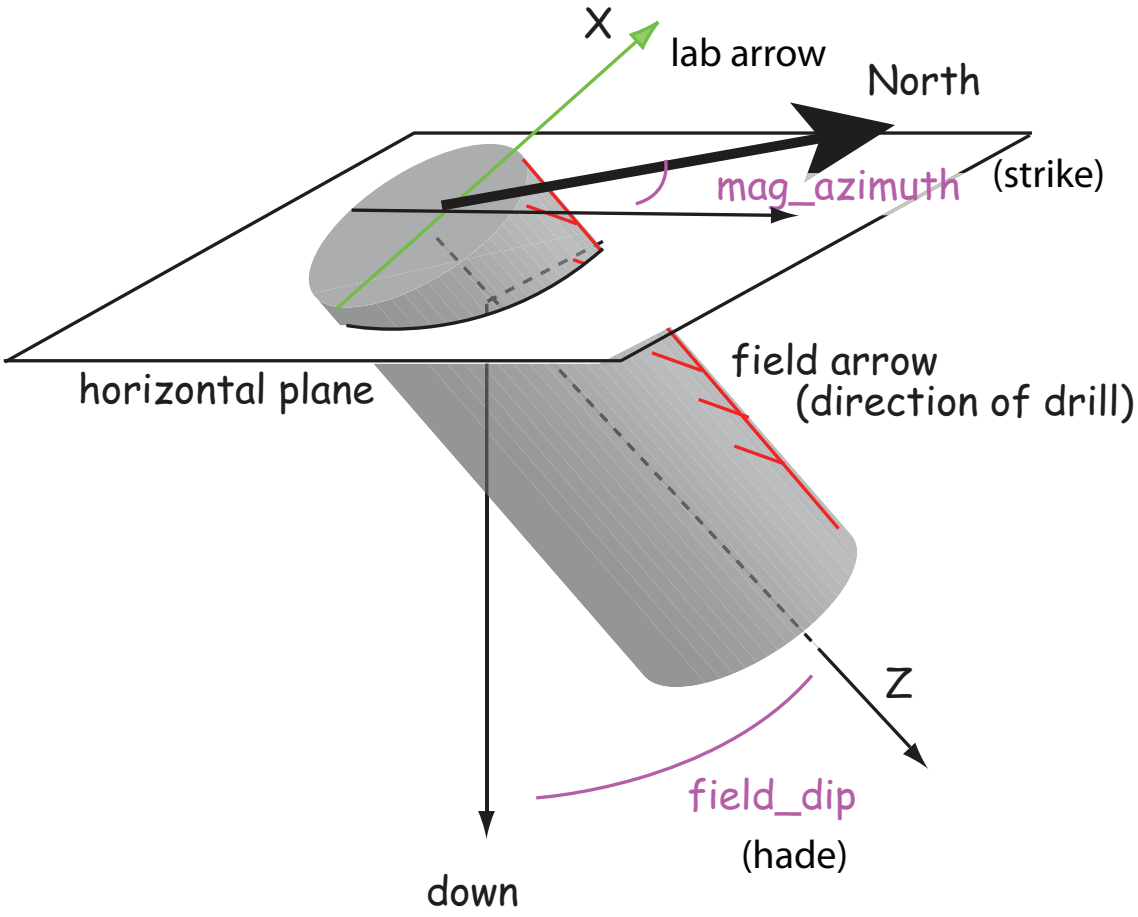
Samples are oriented in the field with a "field arrow" and measured in the laboratory with a "lab arrow". The lab arrow is the positive X direction of the right handed coordinate system of the specimen measurements. The lab and field arrows may not be the same. In the MagIC database, we require the orientation (azimuth and plunge) of the X direction of the measurements (lab arrow). Here are some popular conventions that convert the field arrow azimuth (mag_azimuth in the orient.txt file) and dip (field_dip in orient.txt)

to the azimuth and plunge of the laboratory arrow (sample_azimuth and sample_dip in er_samples.txt). The two angles, mag_azimuth and field_dip are explained below.

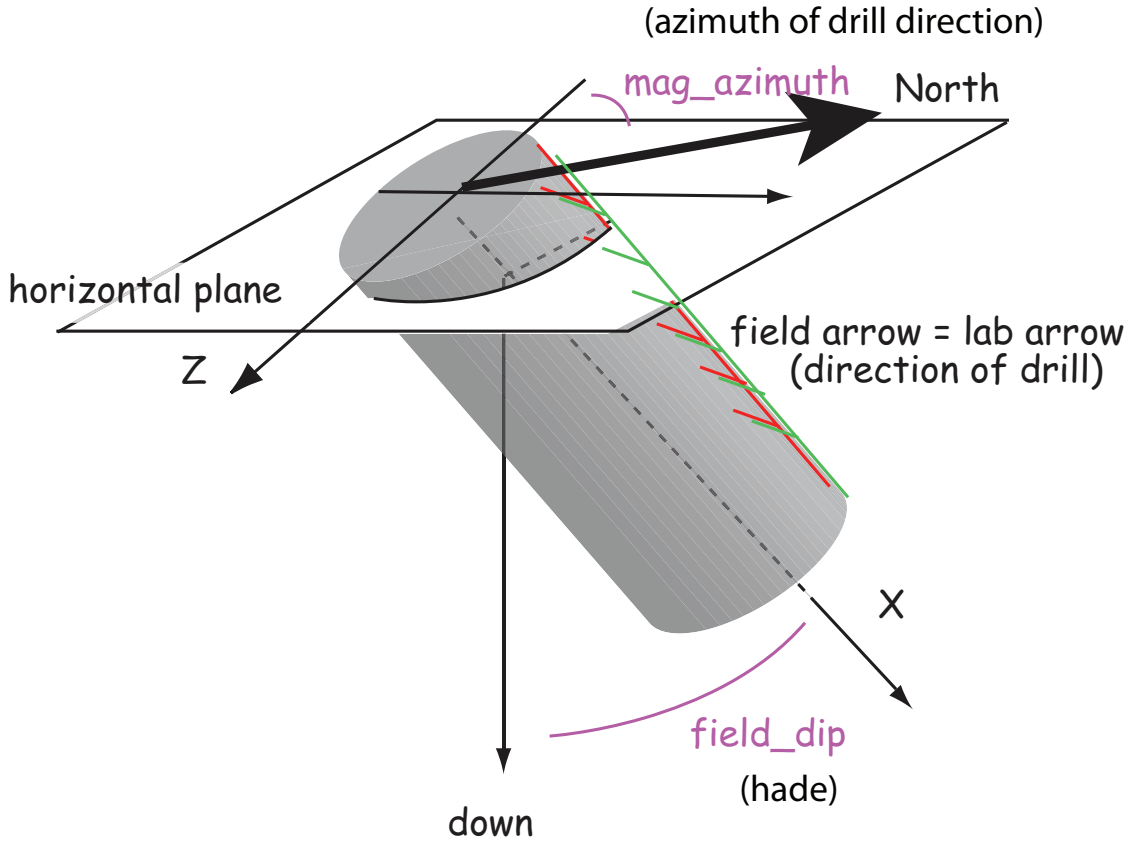
[1] Standard Pomeroy convention of azimuth and hade (degrees from vertical down) of the drill direction (field arrow). sample_azimuth = mag_azimuth; sample_dip =-field_dip.



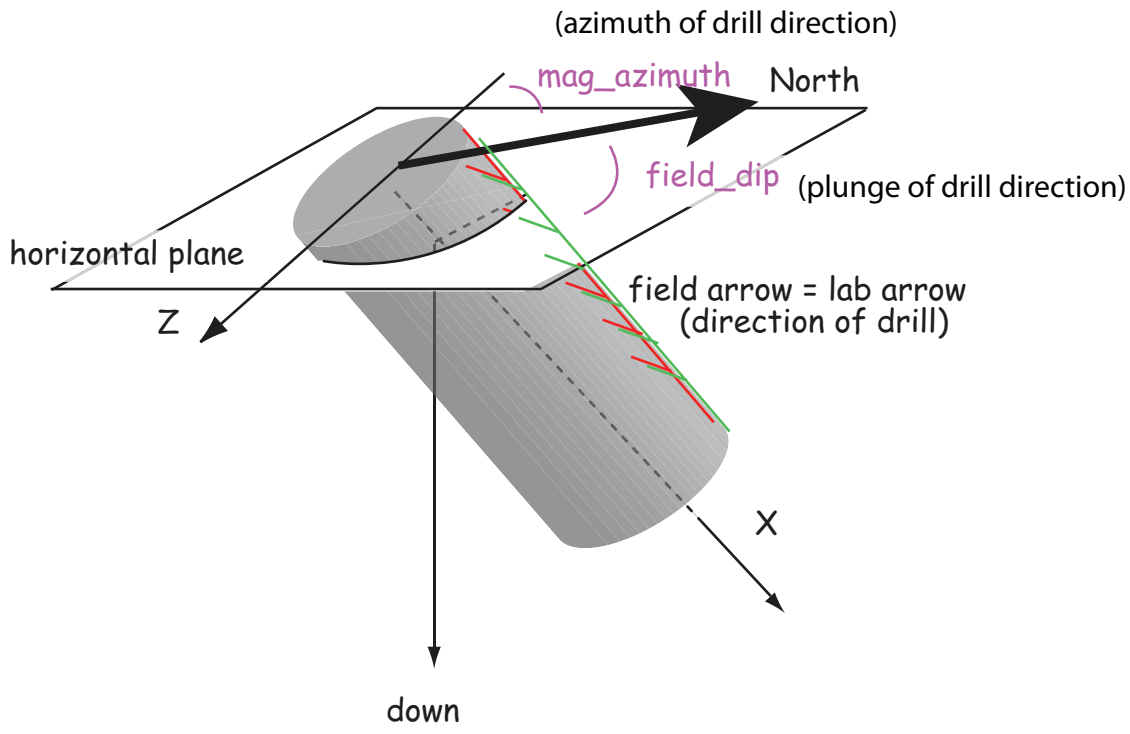
2] Field arrow is the strike of the plane orthogonal to the drill direction, Field dip is the hade of the drill direction. Lab arrow azimuth = mag_azimuth-90°; Lab arrow dip = -field_dip



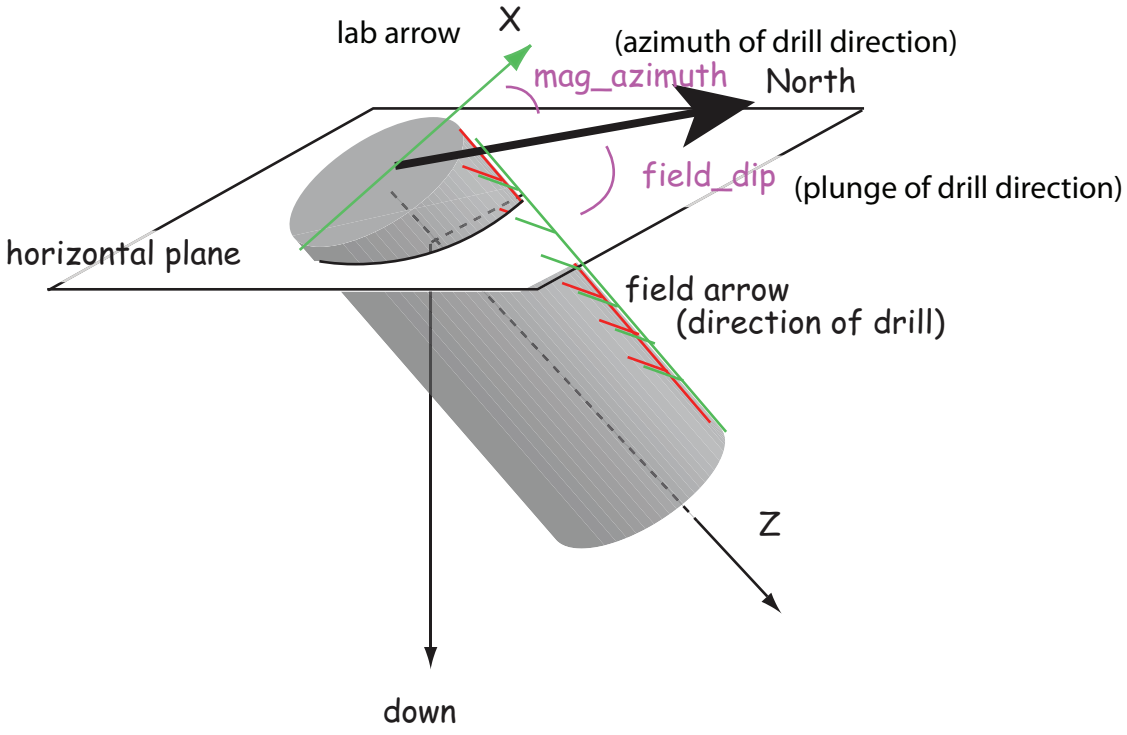
[3] Lab arrow is the same as the drill direction; hade was measured in the field. Lab arrow azimuth = mag_azimuth; Lab arrow dip = 90°-field_dip.



[4] Lab arrow orientation same as mag_azimuth and field_dip.

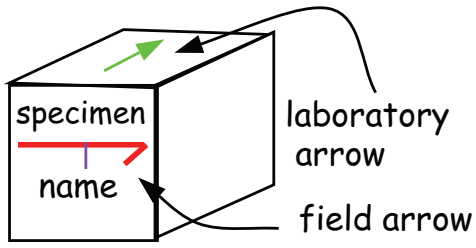


[5] Lab arrow azimuth is mag_azimuth and lab arrow dip is the field_dip-90°



[6] Lab arrow azimuth is mag_azimuth-90°, Lab arrow dip is 90°-field_dip, i.e., the field arrow was strike and dip of orthogonal face:

sample in the field => specimen in the lab



mag_azimuth = strike
field_dip= dip

Structural correction conventions:

Because of the ambiguity of strike and dip, the MagIC database uses the dip direction and dip where dip is positive from 0 → 180. Dips>90 are

overturned beds.

Supported sample naming schemes:

```
[1] XXXXY: where XXXX is an arbitrary length site designation and Y
    is the single character sample designation.  e.g., TG001a is the
    first sample from site TG001.    [default]
[2] XXXX-YY: YY sample from site XXXX (XXX, YY of arbitary length)
[3] XXXX.YY: YY sample from site XXXX (XXX, YY of arbitary length)
[4-Z] XXXX[YYY]:  YYY is sample designation with Z characters from si
[5] site name = sample name
[6] site name entered in site_name column in the orient.txt format in
[7-Z] [XXX]YYY:  XXX is site designation with Z characters from sampl
```

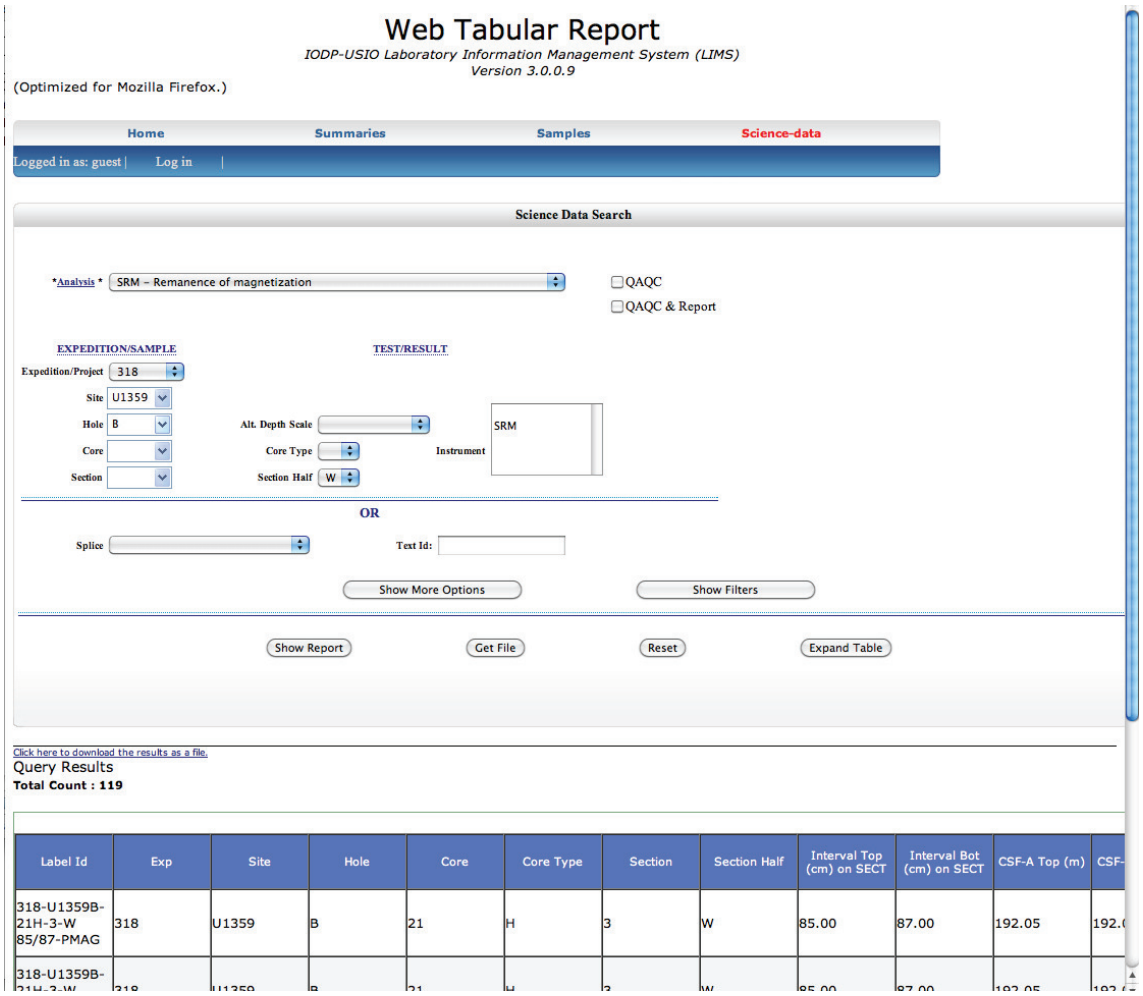
AZDIP formatted files

This is a very simple file format with the sample name Azimuth Plunge Strike Dip where the Azimuth and Plunge are of the drill direction (Specimen's Z direction) or orientation convention #3 above to convert to the MagIC standard. To convert strike to bedding dip direction, we would just add 90°. Here is an example AzDip file:


```
pp187e          336.4   25.0 304.3   56.7
HP002a          198.6   52.0 000.0    0.0
HP003a          202.7   50.0 000.0    0.0
HP004a          197.2   61.0 000.0    0.0
HP005a          224.0   54.0 000.0    0.0
HP006a          207.7   46.0 000.0    0.0
HP007a          184.7   37.0 000.0    0.0
HP008a          200.4   41.0 000.0    0.0
HP009a          189.9   33.0 000.0    0.0
HP010a          260.4   21.0 000.0    0.0
```

Data files downloaded from the IODP (LIMS) database

There are two types of files that help in plotting of IODP paleomagnetic data sets: the core summaries with depth to core top information and the sample information that contains lists of samples taken. Visiting the IODP science query website at `http://web.iodp.tamu.edu/WTR/html/sci-data.html` allows you to select 'SRM - Remanence of magnetization' under the Analysis scroll down menu. By picking the expedition, site, hole, etc. you can download a .csv format (comma separated values) for the expedition data. (Be aware that this is the rawest form of the data, including disturbed intervals, bad measurements, core ends, etc. and may not be exactly what ended up getting published!). First click on the "Show Report" button, then, "Expand Table", then "Get File":



This can take a very long time, so get yourself a cup of tea.

You can also (while you're at it) click on the 'Summaries' tab and download the coring summaries:

Place both of the downloaded files in your *MyFiles* directory.

Supported Rock Magnetometer files

The MagIC database is designed to accept data from a wide variety of paleomagnetic and rock magnetic experiments. Because of this the magic_measurements table is very complicated. Each measurement only makes sense in the context of what happened to the specimen before measurement and under what conditions the measurement was made (temperature, frequency, applied field, specimen orientation, etc). Also, there are many different kinds of instruments in common use, including rock magnetometers, susceptibility meters, Curie balances, vibrating sample and alternating gradient force magnetometers, and so on. We have made an effort to write translation programs for the most popular instrument and file formats and continue to add new supported formats as the opportunity arises. Here we describe the various supported data types and tell you how to prepare your files for importing. In general, all files for importing should be placed in the MyFiles directory or in subdirectories therein as needed. If you don't see your data type in this list, please send an example file and a request to: ltauxe@ucsd.edu and we'll get it in there for you.

The supported file formats are:

Rock Magnetometer Files:

- CalTech's format for Craig Jone's PaleoMag software
- Hebrew University, Jerusalem format
- Scripps Institution of Oceanography format
- Lamont Doherty Earth Observatory format
- IODP rock magnetometer (SRM) data downloaded from the LIMS database
- PMD (ascii) format
- ThellierTool format

Anisotropy of Magnetic Susceptibility files:

- The 6 tensor element format (.s)
- LabView program of Gee et al. 2008

- 15 measurement convention of Tauxe 1998
- SUFAR 4.0 ascii output for Kappabridge

Hysteresis file formats

MagIC.py will import hysteresis data from room temperature Micromag alternating gradient magnetometers (AGM) in several different ways. You can import either hysteresis loops or backfield curves, or you can import whole directories of the same. In the latter case, the file endings must be either .agm (.AGM) or .irm (.IRM) and the first part of the file must be the specimen name. See the documentation for AGM_magic.py for examples.

Prior interpretations

If you already have a PmagPy "redo" file, you can import it into the MagIC folder with this option.

Many investigators are used to their own data analysis routines (e.g., LSQ and PMM). These are not supported at this time. If you have example files and want to see them supported within MagIC.py, please send them to me at: ltauxe@ucsd.edu.

Now you've collected together all the files you need, we can start importing them into *MagIC* directory with **MagIC.py**.

Import MagIC formatted file

There are a number of other datafiles necessary for uploading into the MagIC database, such as the *er_locations.txt, er_ages.txt, er_citations.txt*, etc. files. These tab delimited text files can be created with Excel (or other program) and imported into your MagIC folder with this option.

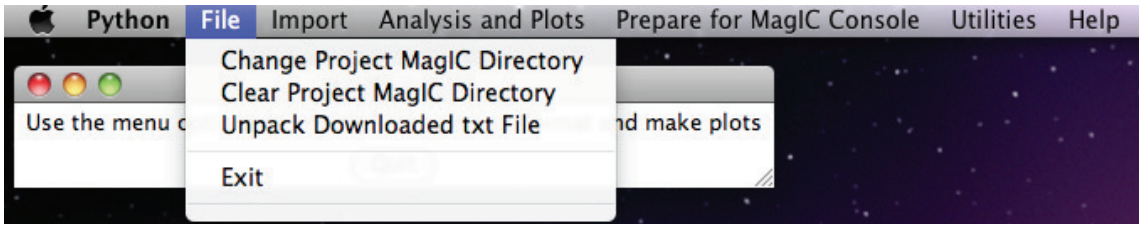
4.2 Using the MagIC.py GUI

The MagIC.py graphical user interface is a Python program that facilitates importing of measurement data and sample information (location, orientation, etc.) into the MagIC format and interpretation thereof. It will help prepare all the files into a text format that can be imported directly into a MagIC smartbook. MagIC.py copies files to be uploaded into a special project MagIC directory, translates them into the MagIC format and keeps track of things in various log files. Note that when it does this, it will translate them into a Unix file format (from MacOS and Windows file formats)

so you don't have to. Once the MagIC project directory has been created, you should just leave it alone. Assuming you have placed all the needed files (orient.txt formatted files for each location and the measurement data files) in the *MyFiles* directory, open up a terminal window and type MagIC.py on the command line. Select the MagIC directory in your Project Directory when prompted.

4.2.1 The File menu

Different operating systems will have a different look, but all versions will put up a Welcome window when you have fired up the program MagIC.py. When you pull down the "File" menu, you will see these options either on the top of your Desktop (Mac OS) or on the top of the window itself (Windows version):



- Change Project MagIC Directory: Allows you to switch the Project Directory that you are working in without exiting the program.

- Clear Project MagIC Directory: Erases all files in the current directory - be careful with this one!

- Unpack Downloaded txt file: After downloading a text file from the MagIC website, you can unpack the contents into a MagIC project directory and work on it using the PmagPy software using this option. This link calls the program download_magic.py.

- Exit: Quits the MagIC.py program.

4.2.2 The Import Menu

When you pull down the "Import" menu, you will see these options:

Importing field and sampling information

Under the 'Orientation/location/stratigraphic files' menu, you will find:

- orient.txt format: imports orient.txt files by calling the program orientation_magic.py. The GUI first copies your file into the MagIC Project Directory, presents you with a few forms to fill out and then generates the correct switches for the command line call.
  - Select orientation file: choose your orient.txt file from MyFiles
  - Select naming convention: Here you explain the relationship between your sample name and the site name. (see supported naming schemes for more info).
  - Select orientation convention...: Choose the transformation of your field convention to the Lab arrow (the direction of the 'X' arrow used in measurements. Also, choose whether you want to use the IGRF value calculated by PmagPy for the site latitude, longitude and date, or you want to supply your own blanket correction to all data in the file. You can also supply the correct azimuth in the file and perform no further corrections (DEC=0 option). If the GUI finds sun compass information, it will ask you how many hours to ADD to the time given in the file to get to Greenwich Mean Time (GMT).
  - Select method codes: Choose which method codes should be attached to ALL samples. For example, if they were all drilled in the field, then check 'FS-FD'.
  - Update and append...: If the GUI detects an existing *er_samples.txt* file from a previous import, it will ask if you want to overwrite it (select no) or append to it - note that sample names used before will be overwritten in any case.

  The GUI will then generate a call with the appropriate switches and your orientation data will be imported. NOTE: If you have already combined your measurements files into the master *magic_measurements.txt* file AND you have changed site designations using the site_name option in the *orient.txt* file, you should select "Update measurements" to change the site names in that file (these get propagated all the way down the line, so it is important to do this step correctly.

- AzDip format: imports AzDip format files with the azdip_magic.py program. Select naming convention and method codes as before. The

orientation is assumed to be the azimuth and plunge of the samples Z direction. This procedure will only update the *er_samples.txt* file and will not have any location information, so you should convert the er_samples.txt file created into an *orient.txt* file using the method outlined below, file in the required fields and re-import that into the MagIC directory (be sure to update your measurements file.

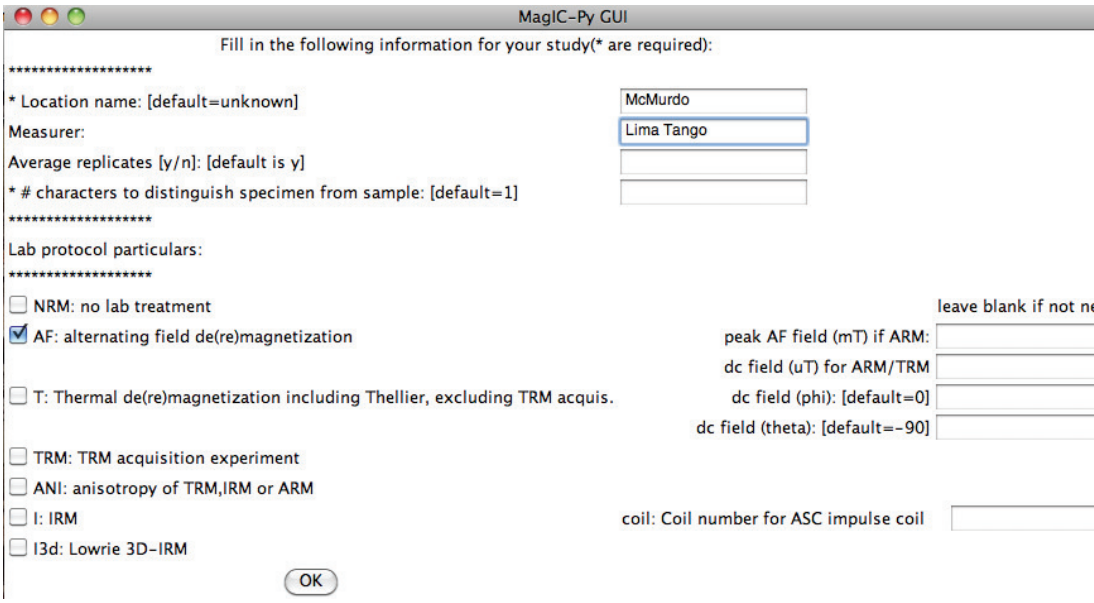
- IODP Core Summary csv file: Imports the IODP core summary files downloaded from the LIMS database website.
- IODP Sample Summary csv file: Imports the IODP sample summary file downloaded from the LIMS database website. This will append to an existing data file, keeping only unique sample names.
- Import model latitude data file: A 'model latitude' is a latitude inferred from either inclination data, or some plate tectonic reconstruction. This file can be used to calculate for example, virtual axial dipole moments (VADMs), or, with the model longitude, a VGP. Use this option to copy a file with the format: er_site_name model_lat model_lon, where the model_lat is the model latitude and the (optional) model_lon is the model longitude. This will be copied into the project directory under the name *model_lat.txt*.

Importing magnetometer data

Under 'Magnetometer files', you will find a variety of file formats described in the section on magnetometer files. Each of these has an idiosyncratic interface. Some allow importing of entire directories at once, while others import data file by file. Be sure to combine your measurements after you finish importing all your measurements, before proceeding to data analysis.

- SIO format: This option will generate a command line call to the program sio_magic.py. There are many options for laboratory procedures, naming conventions and so on and the MagIC.py GUI attempts to walk you through the process with as little pain as possible.
  - Select naming convention: Here you explain the relationship between your sample name and the site name.
  - Fill in the following information: Ylou should really really fill in the location name for this file - it will save you time later if you want to upload your data into the MagIC database. If your specimen/sample relationship is simply to add some characters to te

end of your sample name (for example specimen mc144a1 belongs to sample mc144a), fill in the number of characters you use (unless it is '1' as in this case, because that is the default). If you have some complicated system (for example specimen '##$%&%' belongs to sample 'Harry', I can't help you here and you'll have to fill out all the sample names later by hand. (That should teach you a lesson).



You should also check one of the lab protocols you used.

* If these data were ARMs, check the AF box and fill in the peak AF and dc field values to the right. If the lab field direction was not along -Z, then fill in what it was.
* If the data included TRMs from a Thellier or TRM anisotropy experiment, fill in the DC field and orientation (but not the peak AF field.)
* If the data were for a TRM acquisition experiment, check the TRM box (but not the 'T' box).
* If these data were part of an anisotropy experiment, check the 'ANI' box and the appropriate AF, T, I box for the type of remanence anisotropy experiment it was.
* If the data were IRMs check that box. If the treatment values were in volts on an ASC impulse coil, give the coil number (1,2, or 3) in the appropriate box.

    * If the data were part of a Lowrie 3D-IRM experiment (Lowrie, 1990), then check that box.

- 2G format: This import procedure generates calls to 2G_bin_magic.py. It differs from that for SIO formatted in several ways:
  - Because 2G files contain measurements for single specimens, there is usually a directory with many files. Importing each file individually is painful, so MagIC.py asks for the directory name and imports ALL files with file endings of .dat or .DAT.
  - No lab protocol need be specified.
  - You create an *er_samples.txt* file from the magnetometer file. For that you should specify at least the orientation method used (e.g., magnetic, sun or sighting), otherwise it is assumed that these are unoriented samples. The *er_sample.txt* file is very minimalist, having no lithology information, required for a MagIC database contribution. So after importing the files, you should use the Convert samples => orient.txt option, file in the required information following the instructions in Section 4.1.2, then re-import it into your Project Directory.

- LDEO format: This import procedure is identical to that for SIO formatted files but generates calls for LDEO_magic.py.

- CIT format: This format is the 'CIT' format used by Craig Jone's **PaleoMag** software package (`http://cires.colorado.edu/people/jones.craig/PMag3.html`). **MagIC.py** generates calls to the program CIT_magic.py. The GUI import option behaves in a similar fashion to the SIO format import, but you don't specify lab protocol and you can specify method codes. In the absence of method codes, it is assumed the samples were oriented with magnetic compass.

- HUJI format: This import procedure is identical to that for SIO formatted files but generates calls for HUJI_magic.py.

- IODP format: This is a pretty simple operation. Select the file downloaded from the LIMS database and choose whether you want to average replicate measurements or not. This makes a call to IODP_csv_magic.py to do the dirty work.

- PMD (Enkin) format: This import procedure is identical to that for 2G binary files except it makes calls to PMD_magic.py.

- TDT format: This import procedure is similar to that for SIO formatted files, generating calls to TDT_magic.py. Obviously, you would check the box labelled T and none of the others! If you want to specify the lab field direction (which may not be parallel to -Z as in the SIO lab, you also have to specify the lab field strength in μT, even though it is in the file.

Importing Anisotropy of magnetic susceptibility data

Under the Anisotropy import menu you will find:

- LabView program of Gee et al. 2008: You first select the KLY4S format input file then the naming convention. After that, input your Location name (strongly encouraged), the measurer (optional), the number of characters to distinguish specimen from sample and the instrument used (optional). Combine your measurement files when your are through, to plot versus depth file or as equal area projections (suitable sample orientation required).
- 15 measurement file format of Tauxe 1998: This import procedure is the same as for the KLY4S file format.
- SUFAR 4.0 ascii output for Kappabridge: This import procedure is the same as for the KLY4S file format.

Importing hysteresis data

There are two different modes for importing hysteresis/backfield data: by file or by directory. For the former:

1. Select the file.
2. Select the naming convention.
3. Enter the location name, optional measurer, and instrument name, and the specimen name and number of characters to delineate the specimen from the sample.
4. Indicate whether the file is in cgs or SI units and whether this was an IRM backfield experiment

For importing whole directories: Your file names must follow the convention described in the section on hysteresis file formats. Follow the same

steps as before, but you will not be asked for a specimen name or whether the data are from backfield experiments as these are supplied by the file names.

To combine all the data together, select Combining measurement files.

Combining measurement files

Each time you import a measurement file, you create a MagIC version of the imported data file with the same name, but a .magic appended to the end. These filenames are stored in the file *measurements.log*. After you have completed importing all your measurement files, you must combine them together into a single *magic_measurements.txt* file in order to use the other functions in MagIC.py. Just select this option and everything will be done for you.

Creating an orient.txt file from er_samples

This option makes a call to convert_samples.txt to convert the *er_samples.txt* file created by some import procedures into an *orient.txt* file which can be filled in as desired. It can be re-imported by importing orientation information. After re-importing the orient.txt file, you should update your measurements.

Update measurements:

Update your *magic_measurements.txt* file with new site/location information by calling the program update_measurements.py. This should be done after importing a new *orient.txt* file.

Importing prior interpretations

Many people prefer to use the software package that they are familiar with (e.g., PaleoMac software of Cogné, 2003), Craig Jones' **PaleoMag** software, or even previously processed Pmag (Tauxe, 1998) or PmagPy (Tauxe et al., 2010) softare and have generated files containing favorite interpretations of the paleomagnetic data. MagIC.py can import these interpretation files, allowing the user to skip re-analysis and proceed directly to preparation of files for uploading, or re-interpretation of data files using PmagPy starting with the prior interpretations as a basis. In any case, the following interpretation files are supported (or could be):

- PmagPy redo file: This option copies the files used by zeq_magic_redo.py, thellier_gui.py and thellier_magic_redo.py and then runs them (see documentation in mk_redo.py for file formats. This will overwrite any existing interpretation files (*zeq_specimen.txt* or *thellier_specimens.txt*) files you may have created already, so be careful! Also, this is the main way that the thellier_gui.py shares information with the MagIC.py GUI, so import the redo file generated by the auto-interpreter (e.g., *thellier_interpreter_STDEV-OPT_redo* with this option and you will be able to use them within the MagIC.py GUI.

- DIR: Not supported at this time => if desired send example file to ltauxe@ucsd.edu

- LSQ: Not supported at this time => if desired send example file to ltauxe@ucsd.edultauxe@ucsd.edu

- PMM: Not supported at this time =>if desired send example file to ltauxe@ucsd.edu

Copying over your MagIC formatted files

If you have already some MagIC tables ready, you can import them into your MagIC folder with this option. For example, you might have a previous set of specimen interpretations in a *pmag_specimen.txt* file, or age information (in an *er_ages.txt* file. All age information specific to a give sample or site can be documented in an *er_ages.txt* file (see `http://earthref.org/MAGIC/metadata.htm`). These will be assembled with the respective site/samples in the results table when you assemble your results. Other examples might be a location table (*er_locations.txt*) or a citation table (*er_citations.txt*).

This option will copy your MagIC formatted file into the MagIC Project Directory. Be sure to use the standard file names or the MagIC GUI will ignore them.

4.2.3 Analysis and Plots Menu

After you have imported your various datafiles, you can process them using the tools in the Analysis and Plots menu. All of these options generate and execute command line calls for various PmagPy programs. The **MagIC.py** GUI uses default options because it "knows" the names of the files it has created, saving you from having to figure that out. Many of these options require you to type things in on the command line, such as 'a' to s[a]ve plots,

'q' to quit, or simply return to increment the specimen name. Follow the instructions for the specific program that are printed in the terminal window (see also the documentation in the section on PmagPy programs). Once the program has exited, control will be returned to the MagIC.py GUI.

- Customize Criteria: here you can change your criteria for acceptance or rejection of data.
- Demagnetization Data: calls the program zeq_magic.py. It reads in the *magic_measurements.txt* file created when you combine your imported measurement files. If you have imported orientation information, it will make the plots using the geographic coordinate system.
- Thellier-type experiments: There are two options here. The first calls the program thellier_magic.py and the second calls thellier_gui.py. Both read in the *magic_measurements.txt* file created when you combine your imported measurement files. The **thellier_gui.py** program works a little differently than the other **PmagPy** programs in that the output interpretations are not immediately available to **MagIC.py** when you quit **thellier_gui.py**. To use the interpretations, for example from the auto-interpreter function, you must import the redo file as described under the Import menu. You will find the auto-interpreter redo file in the with the *thellier_interpreter* folder.
- Equal area:
  - Quick look - NRM directions: This option first fishes out all the NRM measurements from the *magic_measurements.txt* file and stores them in a file called *nrm_measurments.txt*. The measurement format is then converted to the *pmag_specimen.txt* format using nrm_specimens_magic.py, which also looks for an *er_samples.txt* file and performs geographic and stratigraphic rotations as available. The GUI will ask for desired coordinate system and level of plotting (by location, site, sample, etc. and then call the program eqarea_magic.py to plot the NRM directions.
  - General remanence directions: You are given a choice of tables from *pmag_specimens.txt* up to *pmag_results.txt*, depending on the desired level. You may also choose from among the available coordinate systems and scope of the plot(s) (e.g., location, site, sample). There are several methods available for estimated confidence bounds for the data set including Fisher, Bingham, Kent or bootstrap. Plots are made with eqarea_magic.py.

- Anisotropy data - equal area plots: makes plots with the aniso_magic.py program. You can choose the scope of the plot and the method for estimating confidence intervals.
- Hysteresis data: calls hysteresis_magic.py to plot the data that you imported and combined into a *magic_measurements.txt* file. It generates a file *rmag_hysteresis.txt* with all the hysteresis parameters calculated by the program. These can be plotted using the 'Hysteresis ratio plots' command.
- Hysteresis ratio plots: This calls dayplot_magic.py to make plots of hysteresis ratios and the like in the *rmag_hysteresis.txt* file created by hysteresis_magic.py.
- IRM acquisition: This calls irmaq_magic.py to make plots of IRM acquisition and back field curves. It uses the *magic_measurements.txt* file created when you combine your imported IRM data.
- 3D-IRM experiment: makes plots of the 3D-IRM experiment of Lowrie (1990), using the program lowrie_magic.py. It uses the *magic_measurements.txt* file created when you combine your imported 3D-IRM data and allows the option to normalize the intensity data by the NRM.
- Remanence versus depth/height: makes plots using a call to core_depthplot.py. The GUI allows customization of the plot with choice of data type (inclination, declination, etc.), time scale, depth scale, and customization of plot symbols. If you have imported a core summary file, you can plot the core tops as well. There is a choice of depth scales with a common depth scale (mcd) as an option.
- Anisotropy data versus depth/height: makes plots using a call to ani_depthplot.py, reading the *rmag_anisotropy.txt* and *magic_measurements.txt* files in the Project Directory.

- Reversals test: performs a bootstrap reversals test with the revtest_magic.py program. You can choose coordinate system and selection criteria.

- Fold test: performs a fold test with foldtest_magic.py and allows choice of selection critieria. Note - it takes a while so be patient and you need to have imported structural corrections.

4.2.4 Prepare for MagIC Console

Once all your data have been imported and interpreted using one of the interpretation tools outlined in the section on Analysis and Plots, you can assemble the results into the various tables expected by MagIC, e.g., the *pmag_specimens, pmag_samples, pmag_sites* and *pmag_results* tables (also various rock magnetic tables, if available).

- Assemble specimens: The various programs for interpreting measurement data like zeq_magic.py or thellier_magic.py produce *pmag_specimens.txt* formatted files with names like *zeq_specimens.txt* or *thellier_specimens.txt*. The directional data should be recalculated for various coordinate systems and the intensity data corrected for anisotropy corrections and the like. All of these data need to be assembled into a single *pmag_specimens.txt* file, which is what this option does.

- Assemble results: After assembling the specimen data into a single *pmag_specimens.txt* file, data must be averaged by sample and/or by site and stored in *pmag_sample.txt* and *pmag_site.txt* tables. The directional data can be converted to VGPs and intensities into V[A]DMs, which are kept in the *pmag_results.txt* table. Study level averages such as paleomagnetic pole positions can be calculated. All of these operations are carried out by the program specimens_results_magic.py.

  - You have a choice of selection criteria (default, existing (created with the customize criteria option, and none).
  - For many applications, data are normally averaged first by sample, then by site and this is the default. But and argument could be made that in certain circumstances (e.g., with intensity data in which each specimen is independent of the other), site averages could be made using all the specimen data together and not averaging by sample. Choose -aD to average directions by site and not by sample and -aI to average intensities by site and not by sample. Also, choose -sam to calculate sample level VGPs and VADMs if desired. Otherwise, the program will do this for site level data onlly.
  - For calculating VDMs, you need oriented inclinations and for VADMs, you need a latitude. The latitude can be the present latitude or a reconstructed paleolatitude (model latitude). To choose the present latitude, select '-lat' or import a model latitude file and choose that option.

  - The -p option lets you look at the data site by site to check if there are terrible errors. If you want to reconsider some of the sample orientations, for example, you should use the option in the Utitlities menu to check the sample orientations for common mistakes.
  - You can just skip the directional (-xD) or intensity (-xI) calculations if they are not relevant (this speeds things up a bit).
  - If you want to calculate normal and reverse means for the study, select -pol.
  - You can choose which coordinate systems you want to include in the MagIC tables. The GUI sniffs through the options and presents them to you.
  - The MagIC database wants all non-synthetic data to have ages associated with them. Site/sample specific ages are entered using the *er_ages.txt* table (imported in the Import er_ages option in the Import Menu, but age bounds for the study must be entered so that specimens_results_magic.py can assign age bounds for every result. The GUI therefore asks for these age bounds. Age units are a controlled vocabulary, so choose from this list: [Ga, ka, Ma, years AD (+/-), Years BP, Years Cal AD (+/-) Years Cal BP]. If you enter nothing, the default age bounds will be 0 to 4.5 Ga.

- Prepare upload txt file: The MagIC Console will import a specially formatted ascii file, parsing all the tables into their correct tabs and allowing you to proceed to the final steps of file preparation for uploading into the database. It will create an *er_locations.txt* file for you based on what is in the *er_sites.txt* file. It will ask you for the missing required information. Then the GUI prepares a file called *upload_dos.txt* which can be directly imported into the MagIC console.

4.2.5 Utilities Menu

- Check sample orientations: This program calls the program site_edit_magic.py. It steps through the data site by site and allows you to evaluate orientation information, testing outliers against specific, common field mistakes (arrow wrong direction, wrong scratch mark, etc.)

- Extract Results to table for publication: After assembling your results, you can extract information from the *pmag_results.txt* file as an tab delimited (excel ready) file or as a (my favorite) LaTex formatted

table. That way, the data in your publication will be identical to the data in the MagIC database (what a concept!). This option allows you to do that using the program pmag_results_extract.py. It will create two files: *Intensities.txt* and *Directions.txt*.

- Map of VGPs: makes plots using vgpmap_magic.py, which reads from the *pmag_results.txt* file (where VGPs are stored). To get VGPs, you need to assemble your results first. If you imported orientation data, you can select the coordinate system. You can also choose whether or not to 'flip' the reverse VGPs and the location of the 'eye' in the orthographic plot.

- Map of site locations: makes plots using basemap_magic.py which reads from the *er_sites.txt* file. Choosing high resolution requires that the high resolution datafile be installed. There are many more options available if you use the program on the command line - this is just for a 'quick look'.

- Make IZZI exp. chart: The IZZI paleointensity experiment (Tauxe and Staudigel, 2004) is a complicated protocol which requires some concentration to get the order of heating steps in the right order. This utility calls chartmaker.py to make a chart to facilitate the laboratory procedure.

- Expected directions/paleolatitudes: calls apwp.py to calculate paleodirectiions and paleolatitudes for a give location at a specified time, after assigning it a tectonic plate.

4.2.6 Help Menu

Takes you to the PmagPy Documentation website:
`http://earthref.org/PmagPy/Docs/PmagPy`.

Chapter 5

The PmagPy software package

The **PmagPy** software package is a comprehensive set of programs for paleomagnetists and rock magnetists. Follow the instructions on the PmagPy home page.

When you type something on your command line, your operating system looks for programs of the same name in special places. These are special "paths" so the directory with your Python scripts has to "be in your path". To inform the operating system of the new directory, you need to "set your path". Follow the instructions on the website: `http://earthref.org/PmagPy/set_path.html` for your particular system.

5.1 General characteristics of PmagPy programs

PmagPy scripts work by calling them on a command line. The python scripts must be placed in a directory that is in your "path". To see if this has been properly done, type **dir_cart.py -h** on the command line and you should get a help message. If you get a "command not found" message, you need to fix your path; check the "installing python" page on the software website. Another possible cause for failure is that somehow, the python scripts are no longer executable. To fix this, change directories into the directory with the scripts, and type the command: chmod a+x *.py

For people who hate command line programs and prefer graphical user interfaces with menus, etc., some of the key programs for interpreting paleomagnetic and rock magnetic data are packaged together in a program called MagIC.py. This can be invoked by typing **MagIC.py** on the command

line. The MagIC.py program generates the desired commands for you, so you do not have to learn UNIX or how to use the command line (except to call the **MagIC.py** program itself). Nonetheless, some understanding of what is actually happening is helpful, because the **MagIC.py** program is more limited than full range of **PmagPy** programs. So, here is a brief introduction to how the **PmagPy** programs work.

All **PmagPy** programs print a help message out if you type: **program_name.py -h** on the command line. Many have an "interactive" option triggered by typing **program_name.py -i**. Many also allow reading from standard input and output. The help message will explain how each particular program functions. There are some common features for the command line options:

- Switches are from one to three characters long, preceded by a '-'.
- The switch '-h' always prints the help message and '-i' allows interactive entry of options.
- Options for command line switches immediately follow the switch. For example: -f INPUT -F OUTPUT will set the input file to INPUT and the output to OUTPUT.
- The switch for input files all start with -f and -F for output files.
- -spc -sam -sit -syn -loc are switches relating to specimens, samples, sites, synthetics and locations respectively.
- Capitalized switches suppress an option (e.g., -A means do not average, while -a means DO average).
- -crd [s,g,t] sets the coordinate system
- -fmt [svg,png,jpg] the default image format.
- -sav saves the plots silently and quits the program

The **PmagPy** scripts call on two special modules, the **pmag** and the **pmagplotlib** modules. These contain most of the calculations and plotting functions.

The source code and the help messages for all programs in the PmagPy package are also available online at: `http://earthref.org/PmagPy/pmagpydocs`

A link to these source code/help menu files is provided for each program listed below with the format [program_name_docs].

5.2 Examples of how to use PmagPy programs

In all examples, the '%' prompt stands for whatever command line prompt you have. Download the package containing example data files from:

`http://earthref.org/PmagPy/Datafiles_2.0.zip`

and unzip the file. Data files for the following examples can be found in the directory with the name of the program (except for measurement input files which are all in the *Measurement_Import* folder.

5.2.1 aarm_magic.py

[Essentials Chapter 13] [MagIC] [aarm_magic docs]

Anisotropy of anhysteretic or other remanence can be converted to a tensor and used to correct natural remanence data for the effects of anisotropy remanence acquisition. For example, directions may be deflected from the geomagnetic field direction or intensities may be biased by strong anisotropies in the magnetic fabric of the specimen. By imparting an anhysteretic or thermal remanence in many specific orientations, the anisotropy of remanence acquisition can be characterized and used for correction. We do this for anisotropy of anhysteretic remanence (AARM) by imparting an ARM in 9, 12 or 15 positions. Each ARM must be preceded by an AF demagnetization step. The 15 positions are shown in the k15_magic.py example.

For the 9 position scheme, **aarm_magic.py** assumes that the AARMs are imparted in positions 1,2,3, 6,7,8, 11,12,13. Someone (a.k.a. Josh Feinberg) has kindly made the measurements and saved them an SIO formatted measurement file named *arm_magic_example.dat* in the datafile directory called *aarm_magic*. Note the special format of these files - the treatment column (column #2) has the position number (1,2,3,6, etc.) followed by either a "00" for the obligatory zero field baseline step or a "10" for the in-field step. These could also be '0' and '1'.

We need to first import these into the magic_measurements format and then calculate the anisotropy tensors. These can then be plotted or used to correct paleointensity or directional data for anisotropy of remanence.

So, first use the program **sio_magic.py** to import the AARM data into the MagIC format. The DC field was 50 μT, the peak AC field was 180 mT, the location was 'Bushveld' and the lab protocol was AF and Anisotropy. The naming convention used Option # 3 (see help menu).

Then use the program aarm_magic.py to calculate the best-fit tensor and write out the MagIC tables: *rmag_anisotropy* and *rmag_results*. These files can be used to correct remanence data in a *pmag_specimens* format table

(e.g, intensity data) for the effects of remanent anisotropy (e.g., using the program thellier_magic_redo.py.

Here is a transcript of a session that works. Note that the **sio_magic.py** command is all on one line (which is what the terminal backslash means).

```
% sio_magic.py -f arm_magic_example.dat -loc Bushveld -LP AF:ANI -F aarm_measurer
      -ncn 3 -ac 180 -dc 50 -1 -1

results put in  aarm_measurements.txt


% aarm_magic.py
specimen tensor elements stored in  ./arm_anisotropy.txt
specimen statistics and eigenparameters stored in  ./aarm_results.txt
```

5.2.2 angle.py

[Essentials Appendix A.3.4] [angle docs]

Use the program **angle.py** to calculate the angle (α) between two directions $D = 350.2, I = 26.8; D = 98.6, I = 67.3$.

```
% angle.py -i
Declination 1: [cntrl-D  to quit] 350.2
Inclination 1: 26.8
Declination 2: 98.6
Inclination 2: 67.3
   72.1
Declination 1: [cntrl-D  to quit] ^D
Good bye
```

[NB: PC users will get a more angry sounding exit message]

You can also use this program by reading in a filename using the '-f' option or from standard input (with <). Try this out with the test file in the *angle* directory (*angle.dat*). First examine the contents of the input file using **"cat"** (or " type" on a DOS prompt). Then use **angle.py** to calculate the angles. You can also save your output in a file *angle.out* with the '-F' option:

```
 % cat angle.dat
    11.2    32.9      6.4   -42.9
```

```
   11.5    63.7     10.5   -55.4
   11.9    31.4    358.1   -71.8
  349.6    36.2    356.3   -45.0
   60.3    63.5     58.9   -56.6
  351.8    37.6     55.0   -45.8
  345.5    53.1     44.8   -26.9
   12.2    20.1     13.6   -54.0
  352.1    37.6      5.1   -39.9
  341.2    53.8     25.1   -61.1
....

 % angle.py -f angle.dat
    75.9
  119.1
  103.7
   81.4
  120.1
  100.9
   95.1
   74.1
   78.4
  120.1
......
.
```

5.2.3 ani_depthplot.py

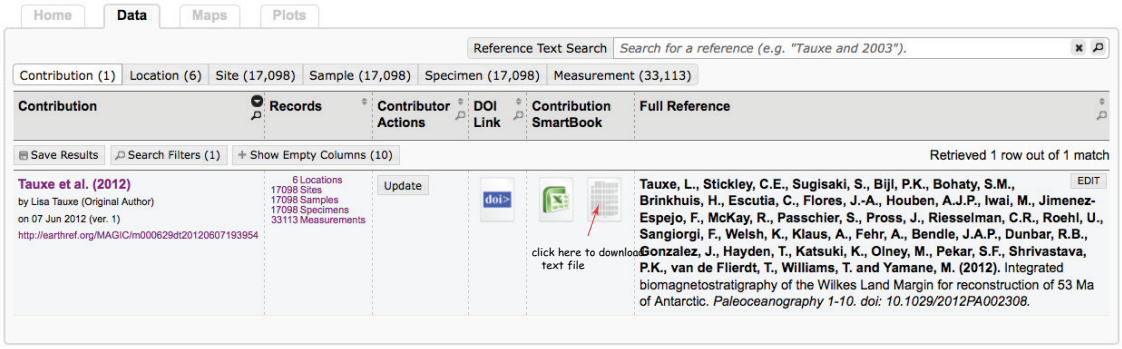
[Essentials Chapter 13]; [MagIC] [ani_depthplot docs]

Anisotropy data can be plotted versus depth. The program **ani_depthplot.py** uses MagIC formatted data tables of the *rmag_anisotropy.txt* and *er_samples.txt* types. *rmag_anisotropy.txt* stores the tensor elements and measurement meta-data while *er_samples.txt* stores the depths, location and other information. Bulk susceptibility measurements can also be plotted if they are available in a *magic_measurements.txt* formatted file.

In this example, we will use the data from Tauxe et al. (2012) measured on samples obtained during Expedition 318 of the International Ocean Drilling Program. To get the entire dataset, go to the MagIC data base at: `http://earthref.org/MAGIC/` and find the data using the search interface. As a short-cut, you can use the "permalink":

`http://earthref.org/MAGIC/m000629dt20120607193954`.

Download the text file by clicking on the icon under the red arrow in:



Unpack the data using the program download_magic.py. This will unpack the data into the different holes. Change directories into Location_2 (which contains the data for Hole U1359A). Or, you can use the data in the **ani_depthplot** directory of the example data files.

```
% ani_depthplot.py
```

will create the plot:

5.2.4 aniso_magic.py

[Essentials Chapter 13]; [MagIC] [aniso_magic docs]

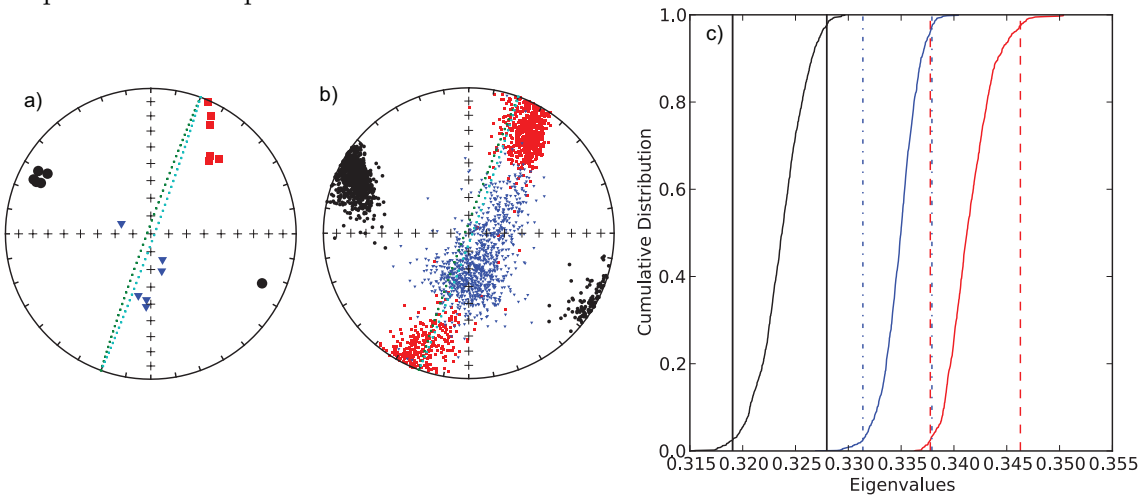
Samples were collected from the eastern margin a dike oriented with a bedding pole declination of 110° and dip of 2°. The data have been imported into a rmag_anisotropy formatted file named *dike_anisotropy.txt*.

Make a plot of the data using **aniso_magic.py**. Use the site parametric bootstrap option and plot out the bootstrapped eigenvectors. Draw on the trace of the dike.

These things are done in this session:

```
% aniso_magic.py -f dike_anisotropy.txt -gtc 110 2 -par -v -crd g
Doing bootstrap - be patient
Boostrap Statistics:
 tau_i, V_i_D, V_i_I, V_i_zeta, V_i_zeta_D, V_i_zeta_I, V_i_eta, V_i_eta_D, V_i_e
0.34040284    29.6    14.5    26.6    165.5     66.3      6.7    295.5     15.7
0.33536589   166.3    70.5    24.6     18.7     17.8     11.1    285.7      9.1
0.32423124   296.2    12.8    12.5    134.6     76.3      6.3     27.3      4.1
compare with [d]irection
 plot [g]reat circle,  change [c]oord. system, change [e]llipse calculation,
     s[a]ve plots, [q]uit
```

which produced these plots:



The specimen eigenvectors are plotted in the left-hand diagram with the usual convention that squares are the V_1 directions, triangles are the V_2 directions and circles are the V_3 directions. All directions are plotted

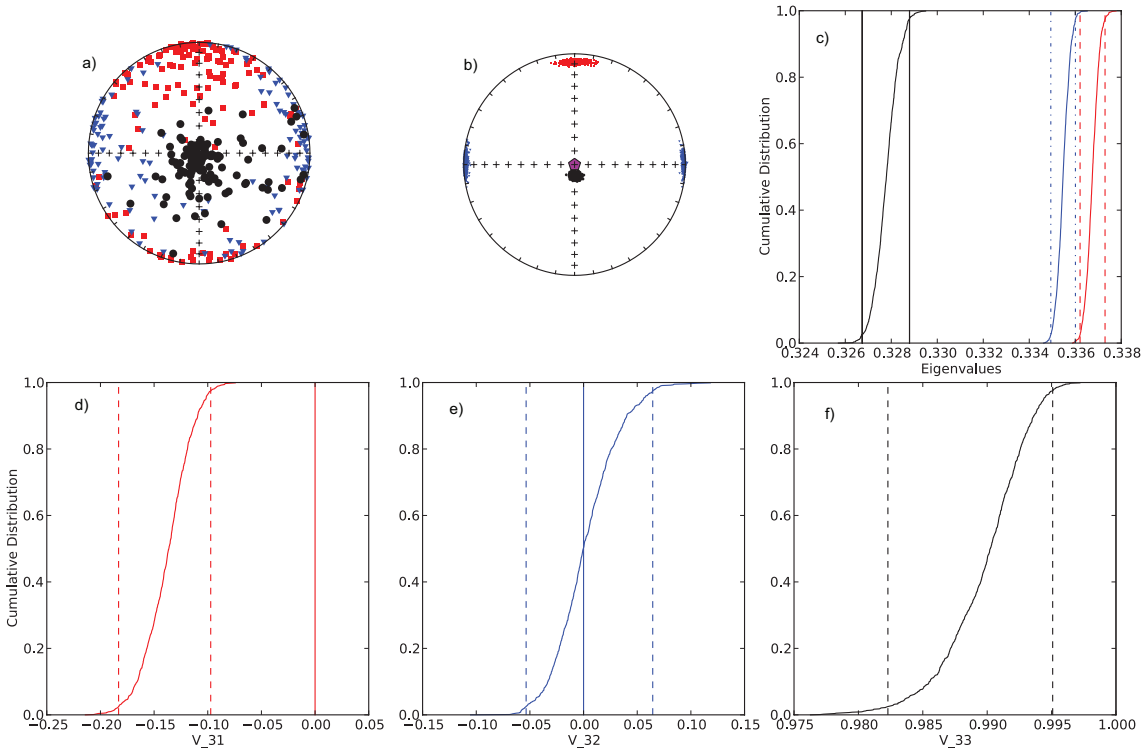
on the lower hemisphere. The bootstrapped eigenvectors are shownin the middle diagram. Cumulative distributions of the bootstrapped eigenvalues are shown to the right with the 95% confidence bounds plotted as vertical lines. It appears that the magma was moving in the northern and slightly up direction along the dike.

There are more options to **aniso_magic.py** that come in handy. In particular, one often wishes to test if a particular fabric is isotropic (the three eigenvalues cannot be distinguished), or if a particular eigenvector is parallel to some direction. For example, undisturbed sedimentary fabrics are oblate (the maximum and intermediate directions cannot be distinguished from one another, but are distinct from the minimum) and the eigenvector associated with the minimum eigenvalue is vertical. These criteria can be tested using the distributions of bootstrapped eigenvalues and eigenvectors.

The following session illustrates how this is done, using the data in the test file *sed_anisotropy.txt* in the aniso_magic directory.

```
%aniso_magic.py -f sed_anisotropy.txt -d 3 0 90 -v -crd g

Doing bootstrap - be patient
Boostrap Statistics:
 tau_i, V_i_D, V_i_I, V_i_zeta, V_i_zeta_D, V_i_zeta_I, V_i_eta, V_i_eta_D, V_i_eta_I
0.33673459   359.5     7.9     3.1    157.5     81.5      9.2    269.1      3.2
0.33546969    89.5     0.0  1349.7    303.3     55.0    153.4    179.7     21.2
0.32779574   179.6    82.1     3.2    358.7      7.9      4.2     88.7      0.1
compare with [d]irection
 plot [g]reat circle,  change [c]oord. system, change [e]llipse calculation,
      s[a]ve plots, [q]uit
```

which makes these plots:



The top three plots are as in the dike example before, showing a clear triaxial fabric (all three eigenvalues and associated eigenvectors are distinct from one another. In the lower three plots we have the distributions of the three components of the chosen axis, V_3, their 95% confidence bounds (dash lines) and the components of the designated direction (solid line). This direction is also shown in the equal area projection above as a red pentagon. The minimum eigenvector is not vertical in this case.

5.2.5 apwp.py

[Essentials Chapter 16] [apwp docs]

The program **apwp.py** calculates paleolatitude, declination, inclination from a pole latitude and longitude based on the paper Besse and Courtillot (2002; see Essentials Chapter 16 for complete discussion). Use it to calculate the expected direction for 100 million year old rocks at a locality in La Jolla Cove (Latitude: 33N, Longitude 117W). Assume that we are on the North American Plate! (Note that there IS no option for the Pacific plate in the program **apwp.py**, and that La Jolla was on the North American plate until a few million years ago (6?).

```
% apwp.py -i

 Welcome to paleolatitude calculator - CTRL-D to quit

pick a plate: NA, SA, AF, IN, EU, AU, ANT, GL

Plate
NA
Site latitude
33
 Site longitude
-117
 Age
100
Age  Paleolat.  Dec.  Inc.  Pole_lat.  Pole_Long.
   100.0    38.8   352.4    58.1    81.5   198.3
```

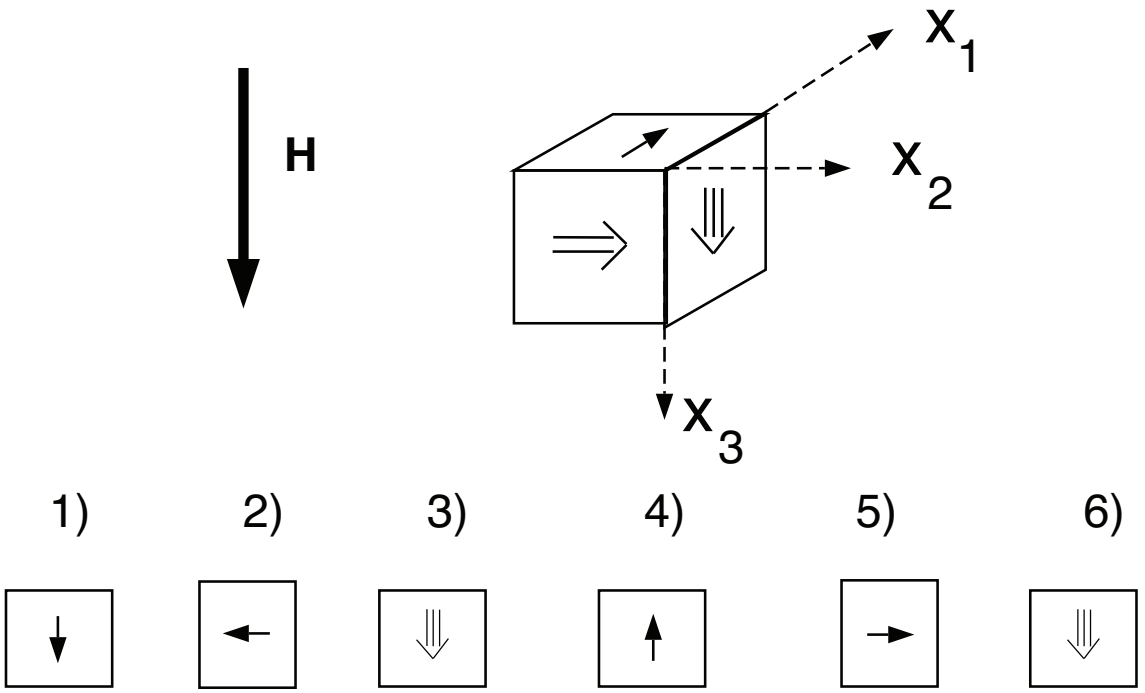
Note that as with many **PmagPy** programs, the input information can be read from a file and the output can be put in a file. For example, we put the same information into a file, *apwp_example.dat* and use this syntax:

```
% apwp.py -f apwp_example.dat
   100.0    38.8   352.4    58.1    81.5   198.3
```

5.2.6 atrm_magic.py

[Essentials Chapter 13] [MagIC] [atrm_magic docs]

Anisotropy of thermal remanence (ATRM) is similar to anisotropy of anhysteretic remanence (AARM) and the procedure for obtaining the tensor is also similar. Therefore, the program **atrm_magic.py** is quite similar to aarm_magic.py. However, the SIO lab procedures for the two experiments are somewhat different. In the ATRM experiment, there is a single, zero field step at the chosen temperature which is used as a baseline. We use only six positions (as opposed to nine for AARM) because of the additional risk of alteration at each temperature step. The positions are also different:



The file *atrm_magic_example.dat* in the *atrm_magic* directory is an SIO formatred data file containing ATRM measurement data done in a temperature of 520°C. Note the special format of these files - the treatment column (column #2) has the temperature in centigrade followed by either a "00" for the obligatory zero field baseline step or a "10" for the first postion, and so on. These could also be '0' and '1', etc..

Use the program sio_magic.py to import the ATRM data into the MagIC format. The DC field was 40 μT. The naming convention used option # 1 (see help menu). Then use the program **atrm_magic.py** to calculate the best-fit tensor and write out the MagIC tables: *rmag_anisotropy* and *rmag_results* formatted files. These files can be used to correct remanence data in a *pmag_specimens* format table (e.g, intensity data) for the effects of remanent anisotropy (e.g., using the program thellier_magic_redo.py.

Here is an example transcript:

```
% sio_magic.py -f atrm_magic_example.dat -loc Africa -LP T:ANI -F atrm_measureme
      -ncn 1  -dc 40 -1 -1

results put in  atrm_measurements.txt
```

```
% atrm_magic.py

specimen tensor elements stored in  ./trm_anisotropy.txt
specimen statistics and eigenparameters stored in  ./atrm_results.txt
```

5.2.7 azdip_magic.py

[Essentials Chapter 9] and [MagIC][azdip_magic docs]

Many paleomagnetists save orientation information in files in this format: Sample Azimuth Plunge Strike Dip (AZDIP format), where the Azimuth and Plunge are the declination and inclination of the drill direction and the strike and dip are the attitude of the sampled unit (with dip to the right of strike). The MagIC database convention is to use the direction of the X coordinate of the specimen measurement system. To convert an *AzDip* formatted file (*example.az*) for samples taken from a location name "Northern Iceland" into the MagIC format and save the information in the MagIC *er_samples.txt* file format, use the program **azdip_magic.py**:

```
% azdip_magic.py -f azdip_magic_example.dat -ncn 1 -mcd FS-FD:SO-POM
      -loc "Northern Iceland"

Data saved in  er_samples.txt
```

Note that there are many options for relating sample names to site names and we used the first convention that has a single character at the end of the site name to designate each sample (e.g., is132d is sample 'd' from site is132). We have also specified certain field sampling and orientation method codes (-mcd), here field sampling-field drilled (FS-FD) and sample orientation-Pomeroy (SO-POM). The location was "Northern Iceland". See the help menu for more options.

Another way to do this is to use the orientation_magic.py program which allows much more information to be imported.

5.2.8 b_vdm.py

[Essentials Chapter 2] [b_vdm docs]

Use the program **b_vdm** to convert an estimated paleofield value of 33 μT obtained from a lava flow at 22° N latitude to the equivalent Virtual Dipole Moment (VDM) in Am^2. Put the input information into a file called *vdm_input.dat* and read from it using standard input :

```
% cat > b_vdm_example.dat
33 22
^D
% b_vdm.py < b_vdm_example.dat
 7.159e+22
```

5.2.9 basemap_magic.py

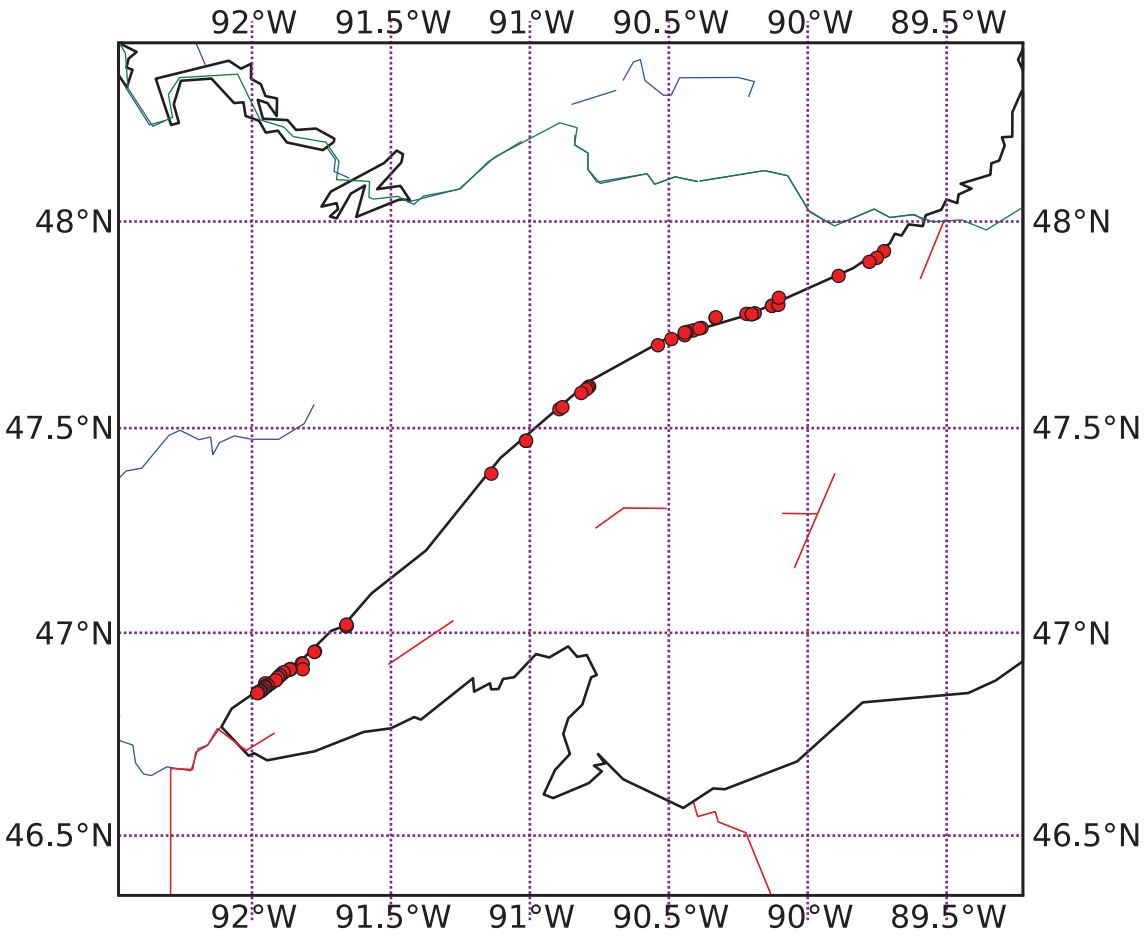
[MagIC] and high resolution instructions [basemap_magic docs]

Python has a complete map plotting package and PmagPy has a utility for making simple base maps for projects. Site location data imported for example using orientation_magic.py into an *er_sites* formatted text file can be plotted using **basemap_magic.py**. There are many options, so check the help message for more details. Note that if you want to use high resolution datafiles or the etopo20 meshgrid (-etp option), you must install the high resolution continental outlines as described on the high resolution instructions website.

As an example, use the program **basemap_magic.py** to make a simple basemap plot with site locations in a MagIC *er_sites.txt* formatted file named *basemap_example.txt*.

```
% basemap_magic.py -f basemap_example.txt
```

which makes this plot:



Use the buttons at the bottom of the plot to resize or save the plot in the desired format.

5.2.10 biplot_magic.py

[Essentials Chapter 8] and [MagIC] [biplot_magic docs]

It is often useful to plot measurements from one experiement against another. For example, rock magnetic studies of sediments often plot the IRM against the ARM or magnetic susceptibility. All of these types of measurements can be imported into a single *magic_measurements* formatted file, using magic method codes and other clues (lab fields, etc.) to differentiate one from another. Data were obtained from a Paleogene core from 28°S for a relative paleointensity study. IRM, ARM, magnetic susceptibility and remanence data were uploaded to the MagIC database. The magic_measurements

formatted file for this study is saved in *core_measurements.txt*.

Use the program **biplot_magic.py** to make a biplot of magnetic susceptibility against ARM. Note that the program makes use of the MagIC method codes which are LT-IRM for IRM, LT-AF-I for ARM (AF demagnetization, in a field), and LP-X for magnetic susceptibility.

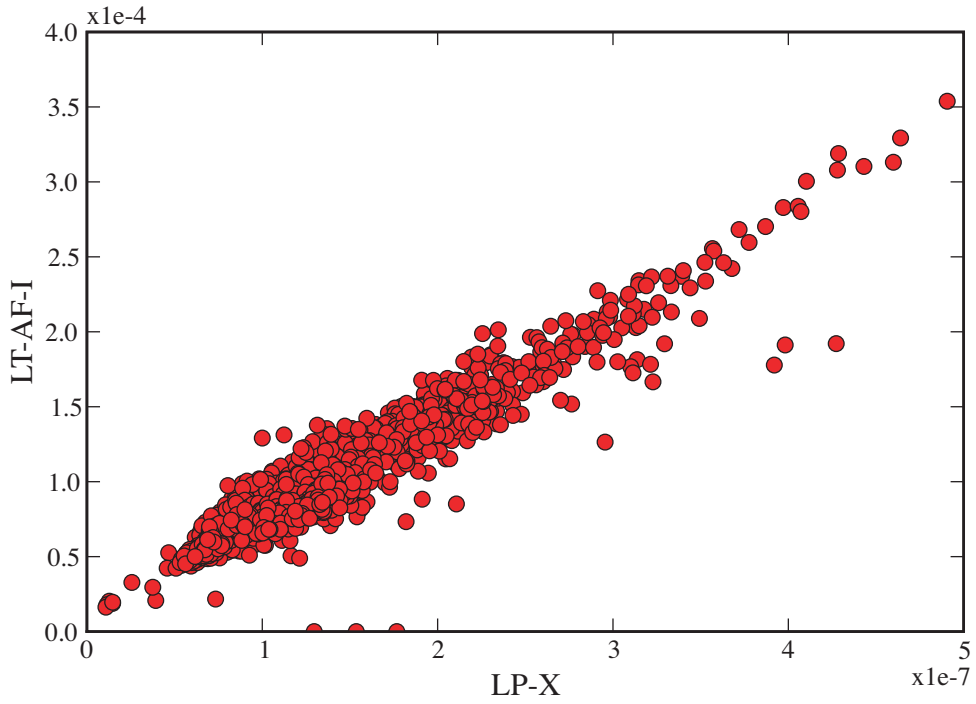
First, to find out which data are available, run the program like this:

```
% biplot_magic.py -f biplot_magic_example.txt
   which responds with this:
   ['LT-AF-Z', 'LT-AF-I', 'LT-IRM', 'LP-X']
```

These are the method codes for AF demagnetization of NRM, ARM, IRM and susceptibility measurements respectively. So to make a plot of susceptibility against ARM, we would run the program again:

```
% biplot_magic.py -f biplot_magic_example.txt -x LP-X -y LT-AF-I
LP-X  selected for X axis
LT-AF-I  selected for Y axis
All
measurement_magn_mass  being used for plotting Y
measurement_chi_mass  being used for plotting X.

S[a]ve plots, [q]uit,  Return for next plot
```

which makes the plot:



5.2.11 bootams.py

[Essentials Chapter 13] [bootams docs]

The program **bootams.py** calculates bootstrap statistics for anisotropy tensor data in the form of:

x11 x22 x33 x12 x23 x13

It does this by selecting para-data sets and calculating the Hext average eigenparameters. It has an optional parametric bootstrap whereby the σ for the data set as a whole is used to draw new para data sets. The bootstrapped eigenparameters are assumed to be Kent distributed and the program calculates Kent error ellipses for each set of eigenvectors. It also estimates the standard deviations of the bootstrapped eigenvalues.

Use this to calculate the bootstrapped error statistics for the data in file *bootams_examples.data*:

```
% bootams.py -par -f bootams_example.dat
Doing bootstrap - be patient
```

```
tau tau_sigma V_dec V_inc V_eta V_eta_dec V_eta_inc V_zeta V_zeta_dec V_zeta_inc

0.33505 0.00021     5.3    14.7    11.7   267.1    28.0    27.0   122.6    56.8
0.33334 0.00023   124.5    61.7    19.6   244.0    14.4    23.5   340.1    22.1
0.33161 0.00026   268.8    23.6    10.4     8.6    21.8    20.4   137.0    57.3
```

Note that every time bootams gets called, the output will be slightly different because this depends on calls to random number generators. If the answers are different by a lot, then the number of bootstrap calculations is too low. The number of bootstraps can be changed with the -nb option.

5.2.12 cart_dir.py

[Essentials Chapter 2] [cart_dir docs]

Use the program **cart_dir.py** to convert these cartesian coordinates to geomagnetic elements:

x_1	x_2	x_3
0.3971	-0.1445	0.9063
-0.5722	0.0400	-0.8192

To use the interactive option:

```
% cart_dir.py -i
X: [cntrl-D  to quit] 0.3971
Y: -0.1445
Z: 0.9063
  340.0    65.0  1.000e+00
X: [cntrl-D  to quit] -.5722
Y: 0.0400
Z: -0.8192
  176.0   -55.0  1.000e+00
```

To read from a file:

```
% cart_dir.py -f cart_dir_example.dat
  340.0    65.0  1.000e+00
  176.0   -55.0  1.000e+00
```

5.2.13 chartmaker.py

[Essentials Chapter 10] [chartmaker docs]

Paleointensity experiments are quite complex and it is easy to make a mistake in the laboratory. The SIO lab uses a simple chart that helps the experimenter keep track of in-field and zero field steps and makes sure that the field gets verified before each run. You can make a chart for an infield-zerofield, zerofield-infield (IZZI) experiment using the program **chartmaker.py**. Make such a chart using 50°C steps up to 500°C followed by 10°C steps up to 600°C.

```
% chartmaker.py
    Welcome to the thellier-thellier experiment automatic chart maker.
    Please select desired step interval and upper bound for which it is valid.
    e.g.,
    50
    500
    10
    600

    a blank entry signals the end of data entry.
    which would generate steps with 50 degree intervals up to 500,
     followed by 10 degree intervals up to 600.

    chart is stored in:  chart.txt

 Enter desired treatment step interval: <return> to quit 50
 Enter upper bound for this interval: 500
 Enter desired treatment step interval: <return> to quit 10
 Enter upper bound for this interval: 600
 Enter desired treatment step interval: <return> to quit
output stored in: chart.txt

%cat chart.txt
file:_________________    field:__________uT


                                date | run# | zone I | zone II | zone III | start | sp |
-------------------------------------------------------------
   0.0               |           |           |              |                |         |
-------------------------------------------------------------
 Z   100.0           |           |           |              |                |         |
-------------------------------------------------------------
```

```
 I   100.1          |          |          |              |                |
-------------------------------------------------------
 T   100.3          |          |          |              |                |
-------------------------------------------------------
 I   150.1          |          |          |              |                |
.
.
.
```

The chart allows you to fill in the file name in which the data were stored and the field value intended for the infield steps. The designations 'Z', 'I', 'T', 'and 'P' are for zero-field, in-field, pTRM tail checks and pTRM checks respectively. There are fields for the date of the runs, the fields measured in different zones in the oven prior to the start of the experiment, and the start and stop times. The numbers, e.g., 100.1 are the treatment temperatures (100) followed by the code for each experiment type. These get entered in the treatment fields in the SIO formatted magnetometer files (see sio_magic.py).

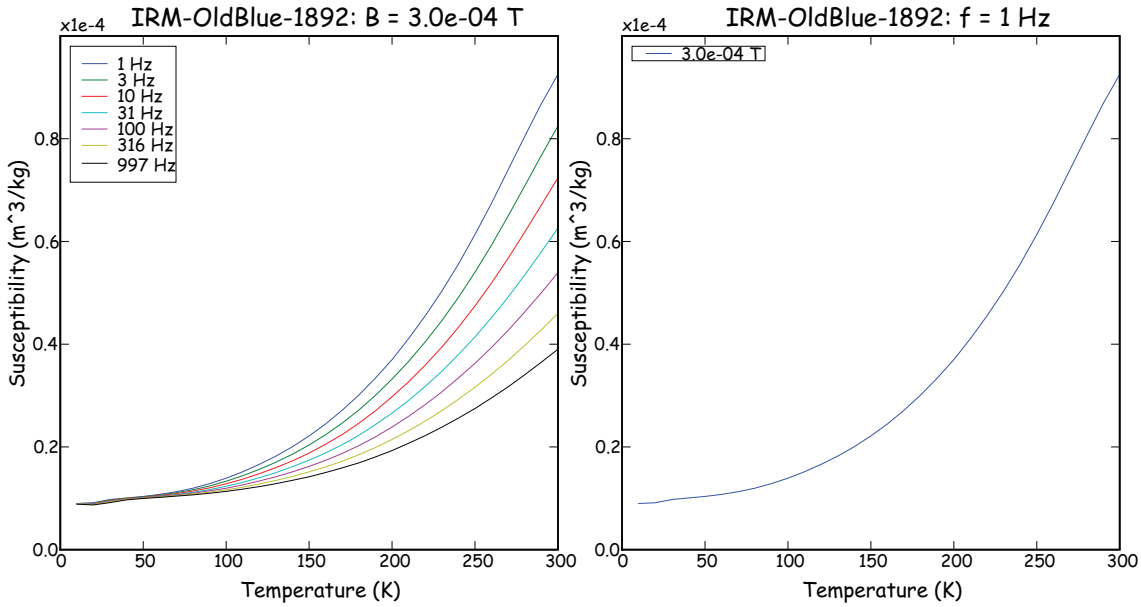
5.2.14 chi_magic.py

[Essentials Chapter 8] & [MagIC] [chi_magic docs]

It is sometimes useful to measure susceptibility as a function of temperature, applied field and frequency. Here we use a data set that came from the Tiva Canyon Tuff sequence (see Carter-Stiglitz, 2006). Use the program **chi_magic.py** to plot the data in the magic_measurements formatted file: *chi_magic_example.dat*.

```
% chi_magic.py -f chi_magic_example.dat
IRM-Kappa-2352 1 out of  2
IRM-OldBlue-1892 2 out of  2
Skipping susceptibitily - AC field plot as a function of temperature
enter s[a]ve to save files,  [return] to quit
```

produced this plot:



You can see the dependence on temperature, frequency and applied field. These data support the suggestion that there is a strong superparamagnetic component in these specimens.

5.2.15 combine_magic.py

[MagIC] [combine_magic docs]

MagIC tables have many columns only some of which are used in a particular instance. So combining files of the same type must be done carefully to ensure that the right data come under the right headings. The program **combine_magic.py** can be used to combine any number of MagIC files from a given type. For an example of how to use this program, see AGM_magic.py.

5.2.16 common_mean.py

[Essentials Chapter 12] [common_mean docs]

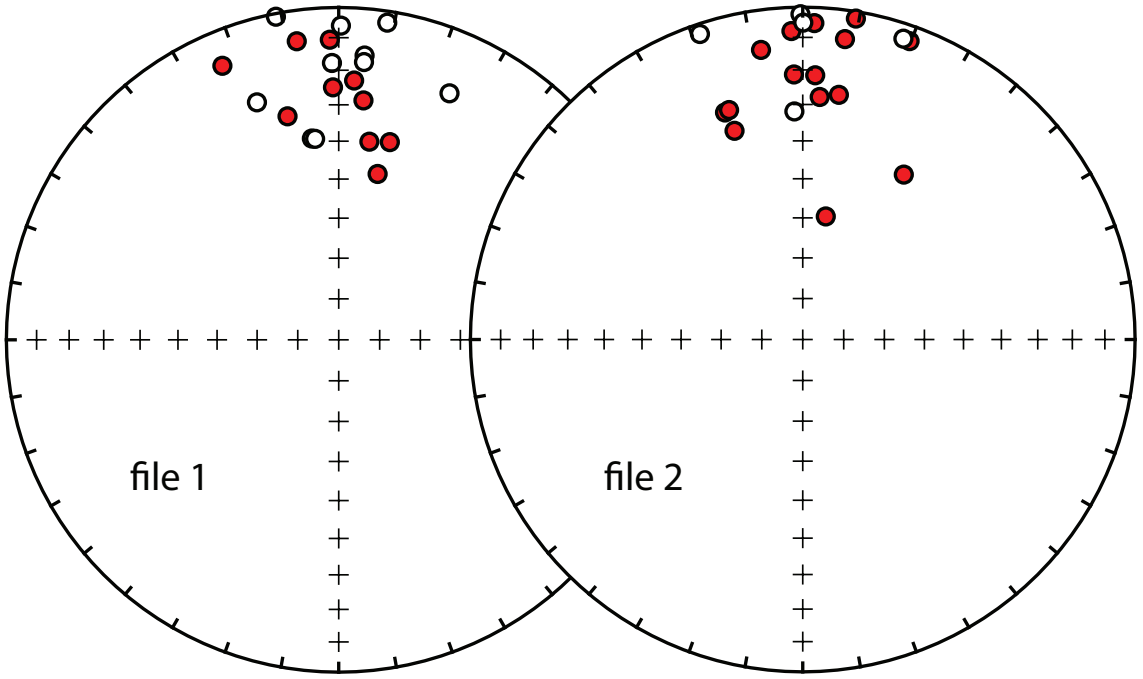
Most paleomagnetists use some form of Fisher Statistics to decide if two directions are statistically distinct or not (see Essentials Chapter 11 for a discussion of those techniques. But often directional data are not fisher distributed and the parametric approach will give misleading answers. In these cases, one can use a boostrap approach, described in detail in [Essentials Chapter 12]. Here we use the program **common_mean.py** for a

bootstrap test for common mean to check whether two declination, inclination data sets have a common mean at the 95% level of confidence. The data sets are: *common_mean_ex_file1.dat* and *common_mean_ex_file2.dat*. But first, let's look at the data in equal area projection using the program eqarea.py.

The session:

```
% eqarea.py -f common_mean_ex_file1.dat -p
1  saved in  common_mean_ex_file1.dat.svg

%  eqarea.py -f common_mean_ex_file2.dat -p
1  saved in  common_mean_ex_file2.dat.svg
```

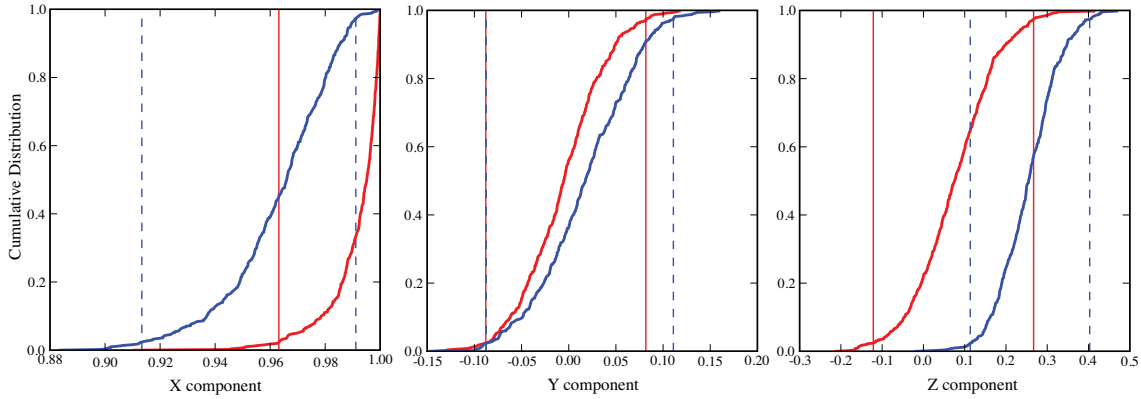
generates two .svg formatted files that look like these:



Now let's look at the common mean problem using **common_mean.py**.

```
% common_mean.py -f common_mean_ex_file1.dat -f2 common_mean_ex_file2.dat
Doing first set of directions, please be patient..
Doing second set of directions, please be patient..
S[a]ve plots, <Return> to quit
```

The three plots are:

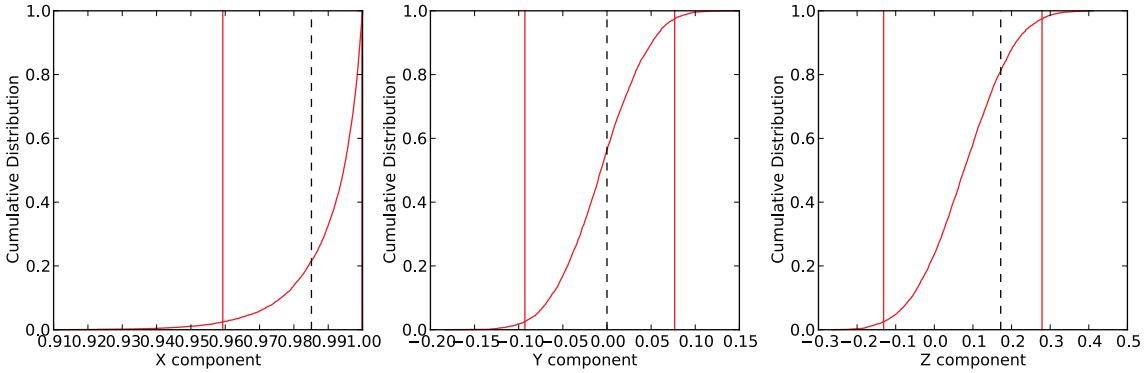


These suggest that the two data sets share a common mean.

Now compare the data in *common_mean_ex_file1.dat* with the expected direction at the 5°N latitude that these data were collected (Dec=0, Inc=9.9).

To do this, follow this transcript:

```
% common_mean.py -f common_mean_ex_file1.dat -dir 0 9.9
Doing first set of directions, please be patient..
S[a]ve plots, <Return> to quit
```



Apparently the data (cumulative distribution functions) are entirely consistent with the expected direction (dashed lines are the cartesian coordinates of that).

5.2.17 cont_rot.py

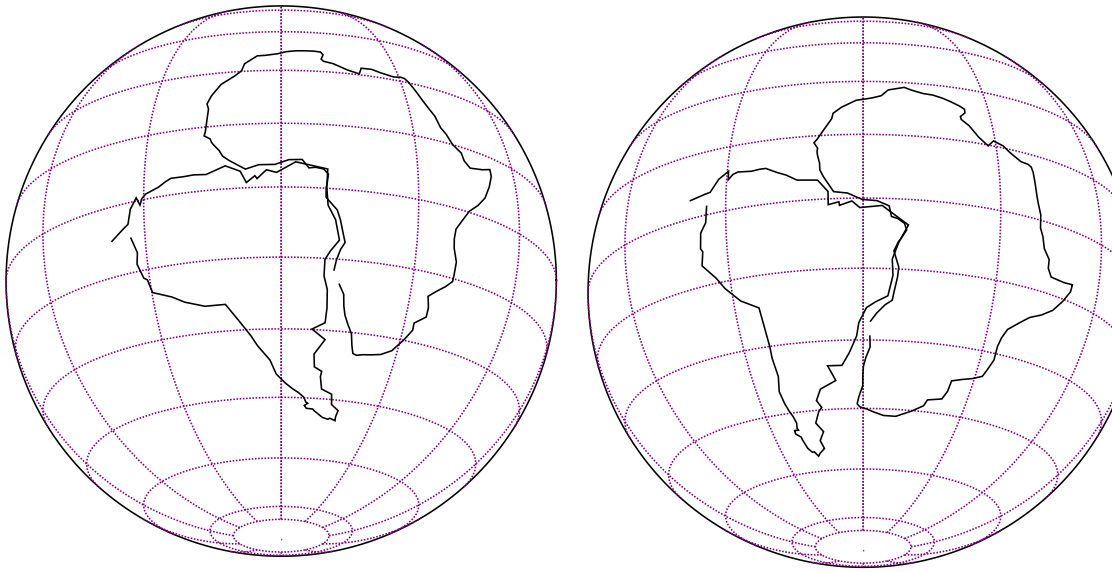
[Essentials Chapter 16 and Essentials Appendix A.3.5.] [cont_rot docs]

Use the program **cont_rot.py** to make an orthographic projection with latitude = -20° and longitude = 0° at the center of the African and South American continents reconstructed to 180 Ma using the Torsvik et al. (2008)

poles of finite rotation. Do this by first holding Africa fixed. Move the output plot to *fixed_africa.svg*. Then make the plot for Africa adjusted to the paleomagnetic reference frame. Make the continental outlines in black lines and set the resolution to 'low'.

```
% cont_rot.py -con af:sam -prj ortho -eye -20 0 -sym k-  1 -age 180 -res l
 S[a]ve to save plot, Return to quit:  a
1  saved in  Cont_rot.pdf
% mv Cont_rot.pdf fixed_africa.pdf
% cont_rot.py -con af:sam -prj ortho -eye -20 0 -sym  k- 1 -age 180 \
     -res l -sac
 S[a]ve to save plot, Return to quit:  a
1  saved in  Cont_rot.pdf
```

These commands generated the following plots (first on left, second on right):



5.2.18 convert2unix.py

[convert2unix docs]

This is a handy little script that turns Windows or Mac file formatted files (not Word or other proprietary formats) into Unix file format. It does the change in place, overwriting the original file.

5.2.19 convert_samples.py

[MagIC] [convert_samples docs]

If one of the MagIC related programs in PmagPy created a very lean looking *er_samples.txt* file (for example using azdip_magic.py) and you need to add more information (say, latitude, longitude, lithology, etc.), you can convert the *er_samples.txt* file into an orient.txt file format, edit it in, for example Excel, and then re-import it back into the *er_samples.txt* file format. Try this on the *er_samples* formatted file in the *convert_samples* directory, *convert_samples_example.dat*.

```
% convert_samples.py -f convert_samples_example.dat
Data saved in:  orient_Northern_Iceland.txt
```

5.2.20 core_depthplot.py

[[Essentials Chapter 15] [core_depthplot docs]

Use the program **core_depthplot.py** to plot various measurement data versus sample depth. The data must be in the MagIC data formats. The program will plot whole core data, discrete sample at a bulk demagnetization step, data from vector demagnetization experiments, and so on. There are many options, so check the help menu before you begin.

We can try this out on some data from DSDP Hole 522, measured by Tauxe and Hartl (1997). These can be downloaded and unpacked (see download_magic.py for details), or you can try it out on the data files in the directory **core_depthplot**. You must specify a lab-protocol (LP) for plotting. In this example, we will plot the alternating field (AF) data after the 15 mT step. The magnetizations will be plotted on a log scale and, as this is a record of the Oligocene, we will plot the Oligocene time scale, using the calibration of Gradstein et al. (2004), commonly referred to as "GTS04" for the the Oligocene. We are only interested in the data between 50 and 150 meters (the -d option sets this) and we will suppress the declinations (-D).

```
% core_depthplot.py -LP AF 15 -log -d 50 150 -ts gts04 23 34 -D
```

will produce the plot:

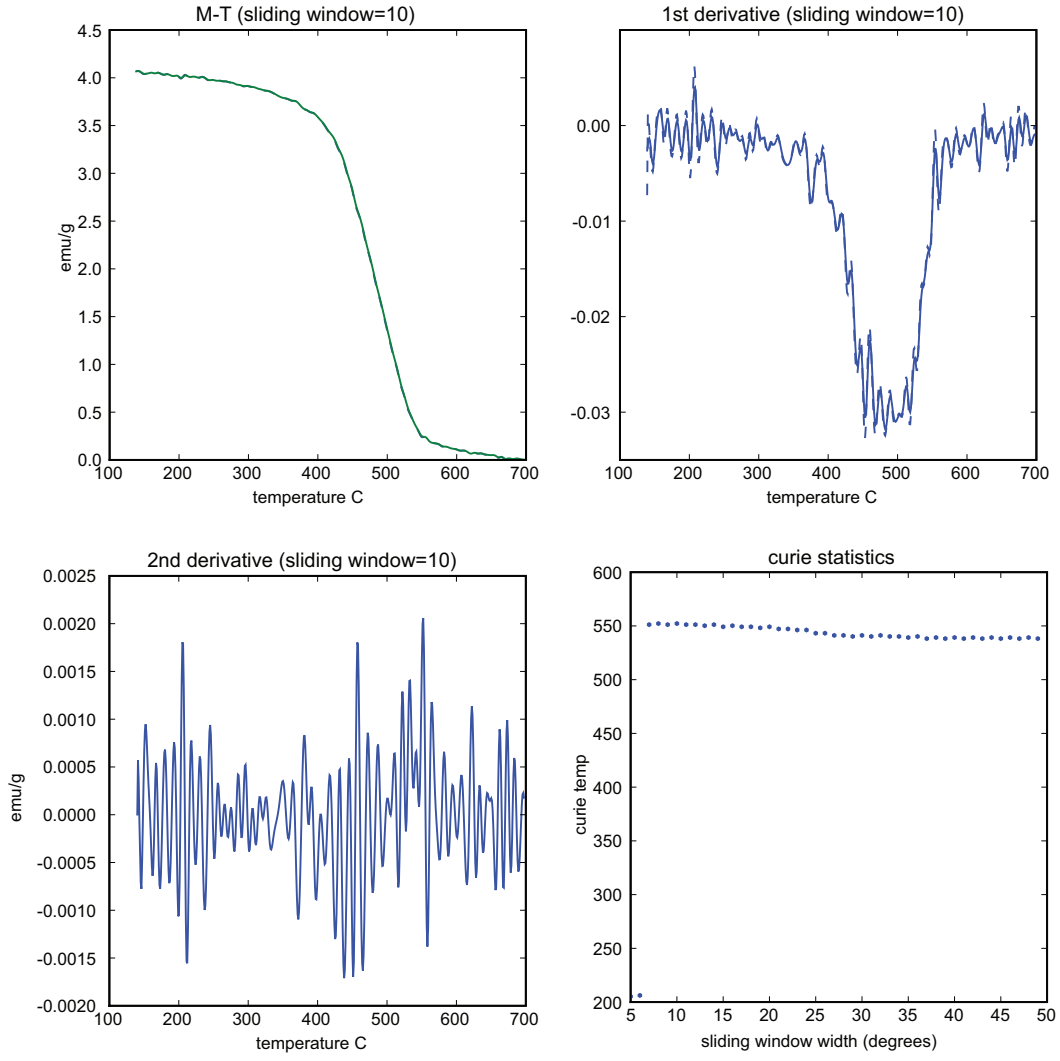
5.2.21 curie.py

[Essentials Chapter 6] [curie docs]

Use the program **curie.py** to interpret curie temperature data in the example file *curie_example.dat*. Use a smoothing window of 10°.

```
%curie.py -f curie_example.txt -w 10
second deriative maximum is at T=552
```

which generates these plots:



5.2.22 customize_criteria.py

[MagIC] [customize_criteria docs]

The MagIC database allows documentation of which criteria were used in selecting data on a specimen, sample or site level and to easily apply those criteria (and re-apply them as they change) in preparing the different tables. These choices are stored in the *pmag_criteria* table in each MagIC project directory (see MagIC.py documentation).

Certain **PmagPy** programs use the datafile *pmag_criteria.txt* to select data (for example **thellier_magic.py** and **specimens_results_magic.py**). To customize these criteria for your own data sets, you can use the program **customize_criteria.py**. This program is also called by the **MagIC.py** GUI under the Utilities menu. Try it out on *pmag_criteria.txt*. This is a "full vector" set of criteria - meaning that it has both directions and intensity flags set. Change the *specimen_alpha95* cutoff to 180. from whatever it is now set to. Save the output to a new file named *new_criteria.txt*.

```
% customize_criteria.py -f pmag_criteria.txt -F new_criteria.txt
Acceptance criteria read in from  pmag_criteria.txt
 [0] Use no acceptance criteria?
 [1] Use default criteria
 [2] customize criteria
 2
Enter new threshold value.
 Return to keep default.
 Leave blank to not use as a criterion

sample_alpha95 10
new value: 5
sample_int_n 2
new value: 2
sample_int_sigma 5e-6
new value:
sample_int_sigma_perc 15
new value:
.......

Criteria saved in pmag_criteria.txt

 Pmag Criteria stored in  pmag_criteria.txt
```

Note that the default place for the PmagPy programs to look for criteria is in *pmag_criteria.txt*, so you should probably rename the new one that for it to take effect as your new default.

5.2.23 dayplot_magic.py

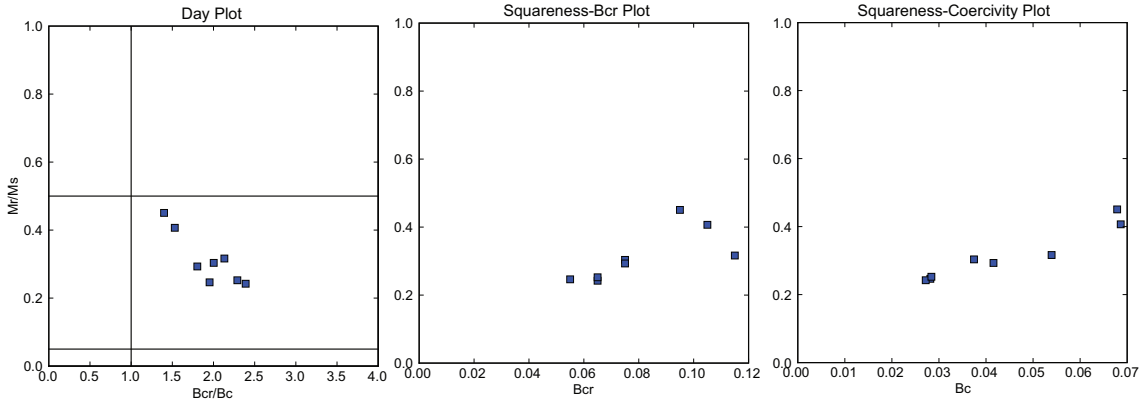
[Essentials Chapter 5] [dayplot_magic docs]

Use the program **dayplot_magic.py** to make Day (Day et al., 1977) , or Squareness-Coercivity and Squareness-Coercivity of Remanence plots (e.g., Tauxe et al., 2002) from the rmag_hyseresis formatted data in *dayplot_magic_example.dat.*

The session:

```
% dayplot_magic.py -f dayplot_magic_example.dat

S[a]ve to save plots, return to quit:
```

gives the plots:



5.2.24 di_eq.py

[Essentials Appendix B] [di_eq docs]

Paleomagnetic data are frequently plotted in equal area projection. PmagPy has several plotting programs which do this (e.g., eqarea.py, but occasionally it is handy to be able to convert the directions to X,Y coordinates directly, without plotting them at all. The program **di_eq.py** does this. Here is an example transcript of a session using the datafile *di_eq_example.dat*:

```
% di_eq.py -f di_eq_example.dat
-0.239410 -0.893491
```

```
0.436413 0.712161
0.063844 0.760300
0.321447 0.686216
0.322720 0.670562
0.407412 0.540654
.
.
.
```

5.2.25 di_geo.py

[Essentials Chapter 9] and Changing coordinate systems [di_geo docs]

Use the programs **di_geo.py** to convert $D = 8.1, I = 45.2$ from specimen coordinates to geographic adjusted coordinates. The orientation of laboratory arrow on the specimen was: azimuth = 347; plunge = 27. **di_geo.py** works in the usual three ways (interactive data entry, command line file specification or from standard input . So for a quickie answer for a single specimen, you can use the interactive mode:

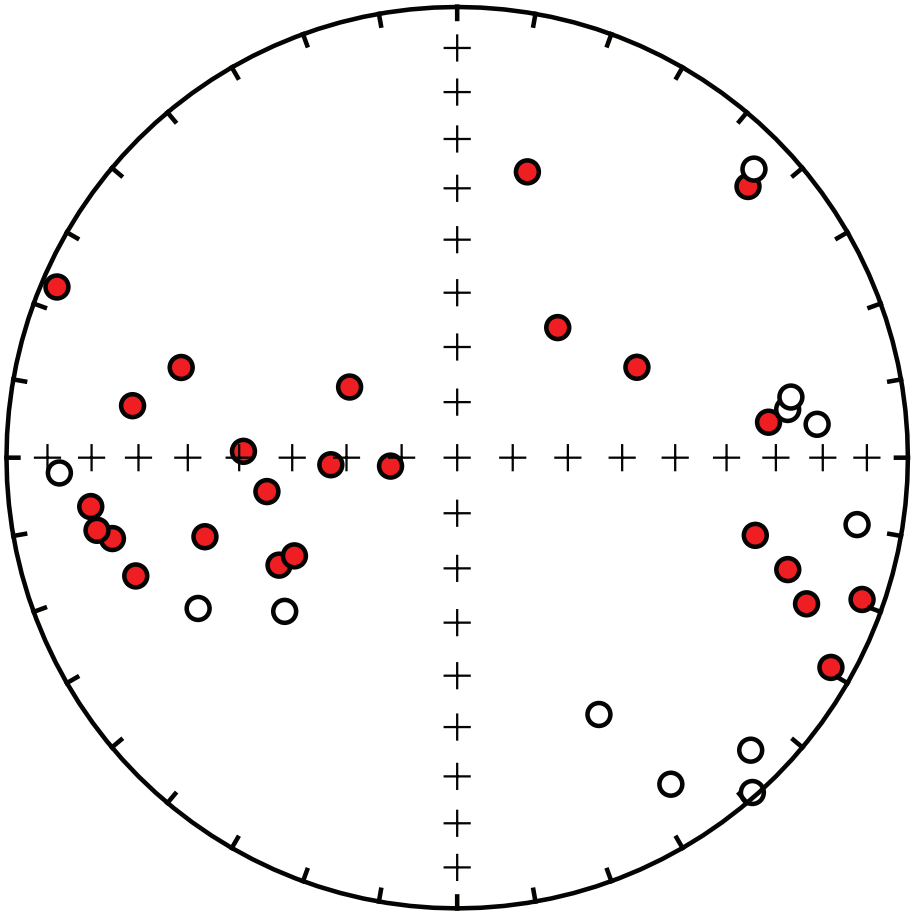
```
% di_geo.py -i
Declination: <cntrl-D> to quit  81
Inclination: 45.2
Azimuth: 347
Plunge: 27
   94.8    43.0
```

which spits out our answer of Declination = 5.3 and inclination = 71.6.

For more data, it is handy to use the file entry options. There are a bunch of declination, inclination, azimuth and plunge data in the file *di_geo_example.dat* in the *di_geo* directory. First look at the data in specimen coordinates in equal area projection, using the program eqarea.py. Note that this program only pays attention to the first two columns so it will ignore the orientation information.

```
% eqarea.py -f di_geo_example.dat
```

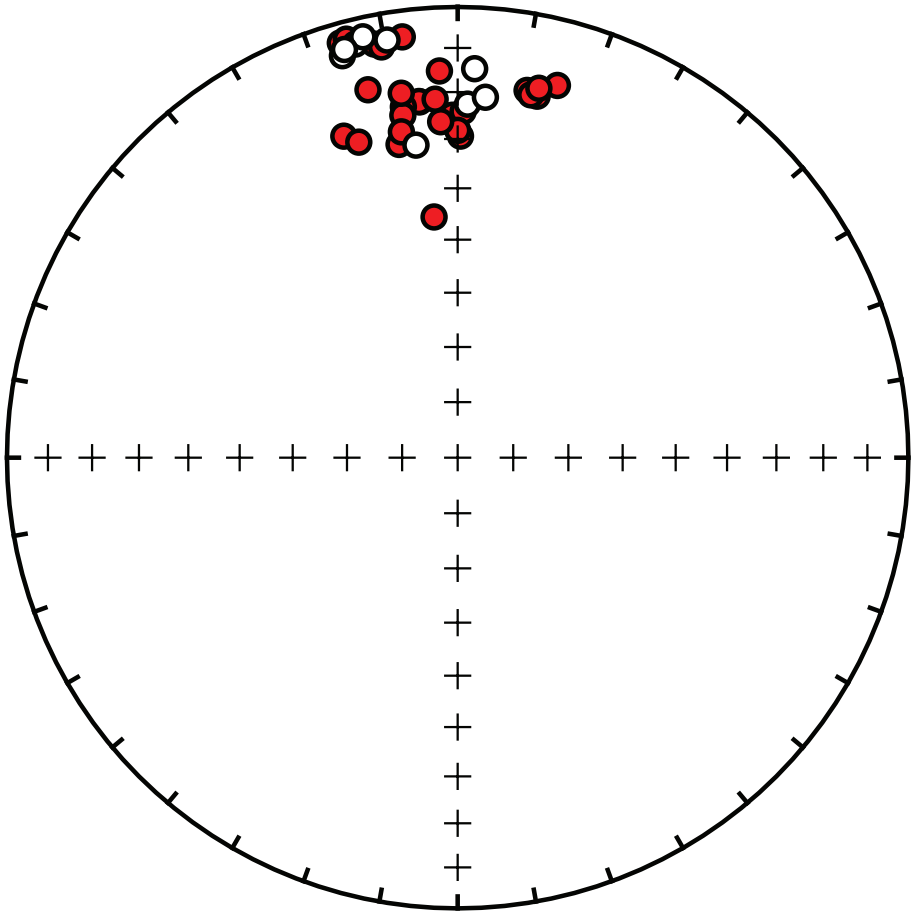
which should look like this:



The data are highly scattered and we hope that the geographic coordinate system looks better! To find out try:

```
di_geo.py -f di_geo_example.dat >di_geo.out;  eqarea.py -f di_geo.out
```

which looks like this:



These data are clearly much better grouped.

5.2.26 di_rot.py

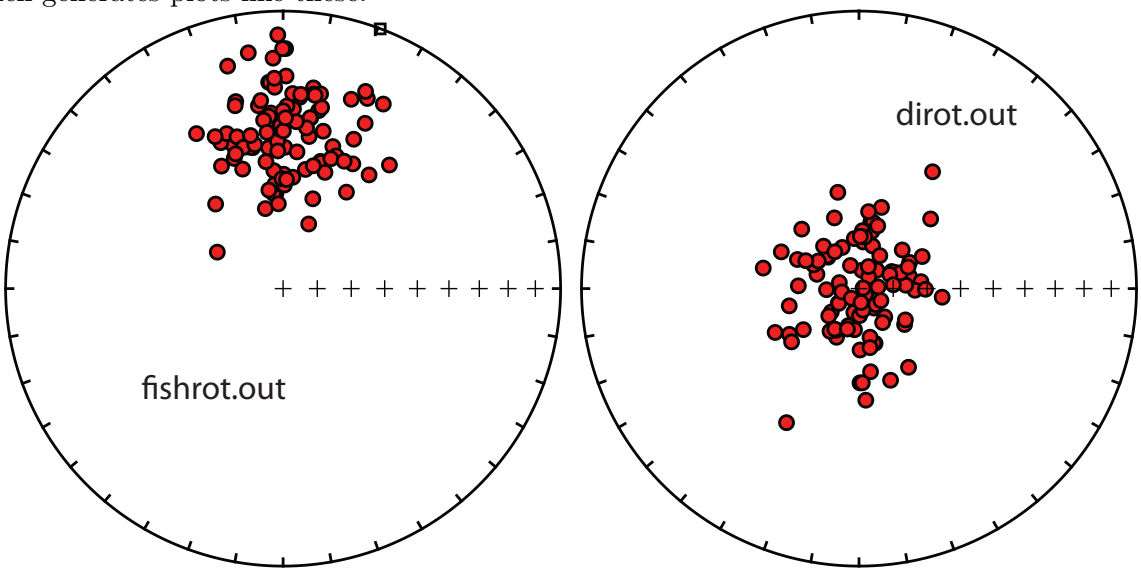
[Essentials Chapter 11] [di_rot docs]

Generate a Fisher distributed set of data from a population with a mean direction of $D = 0, I = 42$ using the program **fishrot.py**. Calculate the mean direction of the data set using **gofish.py**. Now use the program di_rot.py to rotate the set of directions to the mean direction. Look at the data before and after rotation using eqarea.py.

```
% fishrot.py -I 42 >fishrot.out
% gofish.py <fishrot.out
    1.7    42.4    100    95.3720     21.4     3.1
% di_rot.py -f fishrot.out -F dirot.out -D 1.7 -I 42.4
```

```
% eqarea.py -f fishrot.out
 % eqarea.py -f dirot.out
```

which generates plots like these:



Note that every instance of **fisher.py** will draw a different sample from a Fisher distribution and your exact plot and average values will be different in detail every time you try this (and that's part of the fun of statistical simulation.)

5.2.27 di_tilt.py

[Essentials Chapter 9] and Changing coordinate systems [di_tilt docs]

Use the program **di_tilt.py** to rotate a direction of Declination = 5.3 and Inclination = 71.6 to "stratigraphic" coordinates. The strike was 135 and the dip was 21. The convention in this program is to use the dip direction, which is to the "right" of this strike.

Here is a session with **di_tilt.py** using the interactive option:

```
% di_tilt.py -i
Declination: <cntl-D> to quit 5.3
Inclination: 71.6
Dip direction: 225
Dip: 21
  285.7    76.6
Declination: <cntl-D> to quit ^D
```

```
 Good-bye
```

Try the same on the data file saved as *di_tilt_example.dat* using the command line -f switch:

5.2.28 di_vgp.py

[Essentials Chapter 2] [di_vgp docs]

Use the program **di_vgp** to convert the following:

D	I	λ_s (N)	ϕ_s (E)
11	63	55	13
154	-58	45.5	-73

Here is a transcript of a typical session using the command line option for file name entry:

```
% di_vgp.py -f di_vgp_example.dat
  154.7    77.3
    6.6   -69.6
```

5.2.29 dipole_pinc.py

[Essentials Chapter 2] [dipole_pinc docs]

Calculate the expected inclination at a paleolatitude of 24°S.

```
% dipole_pinc.py -i
Paleolat for converting to inclination: <cntl-D> to quit -24
  -41.7
Paleolat for converting to inclination: <cntl-D> to quit ^D
 Good-bye
```

5.2.30 dipole_plat.py

[Essentials Chapter 2] [dipole_plat docs]

Calculate the paleolatitude for an average inclination of 23°.

```
% dipole_plat.py -i
Inclination for converting to paleolatitude: <cntl-D> to quit 23
   12.0
Inclination for converting to paleolatitude: <cntl-D> to quit ^D
 Good-bye
```

5.2.31 dir_cart.py

[Essentials Chapter 2]] [dir_cart docs]

Use the program **dir_cart.py** to convert the following data from declination D, inclination I and intensity M to x_1, x_2, x_3.

D	I	M (μAm^2)
20	46	1.3
175	-24	4.2

You can enter D, I, M data into data file, then running the program by typing what is after the prompts (%) [the other stuff is computer responses] :

```
% cat > dir_cart_example.dat
 20 46 1.3
175 -24 4.2
^D
% dir_cart.py <dir_cart_example.dat
8.4859e-01 3.0886e-01 9.3514e-01
-3.8223e+00 3.3441e-01 -1.7083e+00
```

Or you could use **dir_cart.py** interactively as in:

```
 % dir_cart.py -i

Declination: [cntrl-D  to quit]
 Good-bye

% dir_cart.py -i
Declination: [cntrl-D  to quit] 20
Inclination: 46
Intensity [return for unity]: 1.3
8.4859e-01 3.0886e-01 9.3514e-01
Declination: [cntrl-D  to quit] 175
Inclination: -24
Intensity [return for unity]: 4.2
-3.8223e+00 3.3441e-01 -1.7083e+00
Declination: [cntrl-D  to quit] ^D
 Good-bye
```

5.2.32 dmag_magic.py

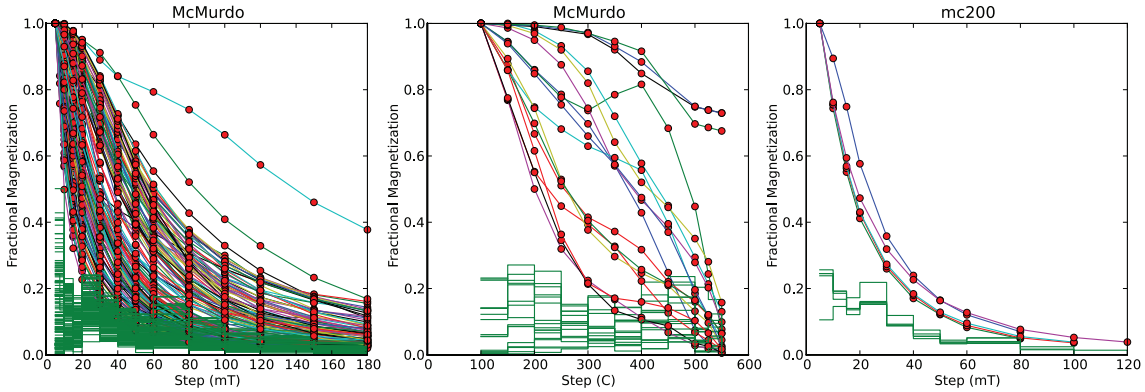
[Essentials Chapter 9] and [MagIC] [dmag_magic docs]

Use **dmag_magic.py** to plot out the decay of all alternating field demagnetization experiments in the magic_measurements formatted file in *dmag_magic_example.dat*. These are from Lawrence et al. (2009). Repeat for all thermal measurements, but exclude all the data acquired during the thermal but not paleointensity experiments. Try this at the location level and then at the site level.

Here is a transcript of a session:

```
%  dmag_magic.py -f dmag_magic_example.dat -LT AF
5921  records read from  dmag_magic_example.dat
McMurdo plotting by:  er_location_name
 S[a]ve to save plot, [q]uit,  Return to continue:  a
1  saved in  McMurdo_LT-AF-Z.svg
% dmag_magic.py -f dmag_magic_example.dat -LT T -XLP PI
5921  records read from  dmag_magic_example.dat
McMurdo plotting by:  er_location_name
 S[a]ve to save plot, [q]uit,  Return to continue:  a
1  saved in  McMurdo_LT-T-Z.svg
% dmag_magic.py -f dmag_magic_example.dat -LT AF -obj sit
5921  records read from  dmag_magic_example.dat
mc20 plotting by:  er_site_name
 S[a]ve to save plot, [q]uit,  Return to continue:
mc200 plotting by:  er_site_name
 S[a]ve to save plot, [q]uit,  Return to continue:  a
1  saved in  mc200_LT-AF-Z.svg
```

which produced these plots:



5.2.33 download_magic.py

[MagIC] [download_magic docs]

This program unpacks .txt files downloaded from the MagIC database into individual directories for each location into which the individual files for each table (e.g., *er_locations.txt, magic_measurements.txt, pmag_results.txt* and so on) get placed. As an example, go to the MagIC data base at `http://earthref.org/MAGIC/search`. Enter "Tauxe and 2004" into the Reference Text Search field will show you several references. Look for the one for Tauxe, L., Luskin, C., Selkin, P., Gans, P. and Calvert, A. (2004). Download the text file under the "Contribution SmartBook" column and save it to your desktop. Make a folder into which you should put the downloaded txt file called MagIC_download and move the file into it. Now use the program **download_magic.py** to unpack the .txt file (*zmab0083201tmp03.txt*).

```
% mkdir MagIC_download
% cd MagIC_download
% download_magic.py -f zmab0083201tmp03.txt
% download_magic.py -f zmab0083201tmp03.txt
working on:  'er_locations'
er_locations  data put in  ./er_locations.txt
working on:  'er_sites'
er_sites  data put in  ./er_sites.txt
working on:  'er_samples'
er_samples  data put in  ./er_samples.txt
working on:  'er_specimens'
.
.
.
location_1:  Snake River
unpacking:  ./er_locations.txt
1  read in
1  stored in  ./Location_1/er_locations.txt
unpacking:  ./er_sites.txt
27  read in
27  stored in  ./Location_1/er_sites.txt
unpacking:  ./er_samples.txt
271  read in
271  stored in  ./Location_1/er_samples.txt
unpacking:  ./er_specimens.txt
```

.
.
.

You can change directories into each Location directory (in this case only one) and examine the data using the **PmagPy** programs (e.g., zeq_magic.py).

5.2.34 eigs_s.py

[Essentials Chapter 13] [eigs_s docs]

Print out the eigenparameters in the file *eigs_s_example.dat* and then convert them to tensor data in the .s format (x11,x22,x33,x12,x13,x23).

This session uses the unix utility **cat** to print the data. [You could use the Ms-Dos form **type** in a Windows command line window.] Then, it prints the tensor data to the screen.

```
% cat eigs_s_example.dat
0.33127  239.53   44.70 0.33351  126.62   21.47 0.33521  19.03   37.54
0.33177  281.12    6.18 0.33218  169.79   73.43 0.33603   12.82   15.32
...
% eigs_s.py -f eigs_s_example.dat
0.33416328 0.33280227 0.33303446 -0.00016631 0.00123163 0.00135521
0.33555713 0.33197427 0.33246869 0.00085685 0.00025266 0.00098151
0.33585301 0.33140355 0.33274350 0.00132308 0.00117787 0.00000455
...
```

5.2.35 eq_di.py

[Essentials Appendix B] [eq_di docs]

Data are frequently published as equal area projections and not listed in data tables. These data can be digitized as x,y data (assuming the outer rim is unity) and converted to approximate directions with the program **eq_di.py**. To use this program, install a graph digitizer (GraphClick from `http://www.arizona-software.ch/graphclick/` works on Macs).

Digitize the data from the equal area projection saved in the file *eqarea.png* in the *eq_di* directory. You should only work on one hemisphere at a time (upper or lower) and save each hemisphere in its own file. Then you can convert the X,Y data to approximate dec and inc data - the quality of the data depends on your care in digitizing and the quality of the figure that you are digitizing.

Try out **eq_di.py** on your datafile, or use *eq_di_example.dat* which are the digitized data from the lower hemisphere and check your work with **eqarea.py**. You should retrieve the lower hemisphere points from the **eqarea.py** example.

```
% eq_di.py -f eq_di_example.dat >tmp
% eqarea.py -f tmp
```

NB: To indicate that your data are UPPER hemisphere (negative inclinations), use the -up switch.

You can verify the process by comparing the plot generated for these data using eqarea.py with the original png file.

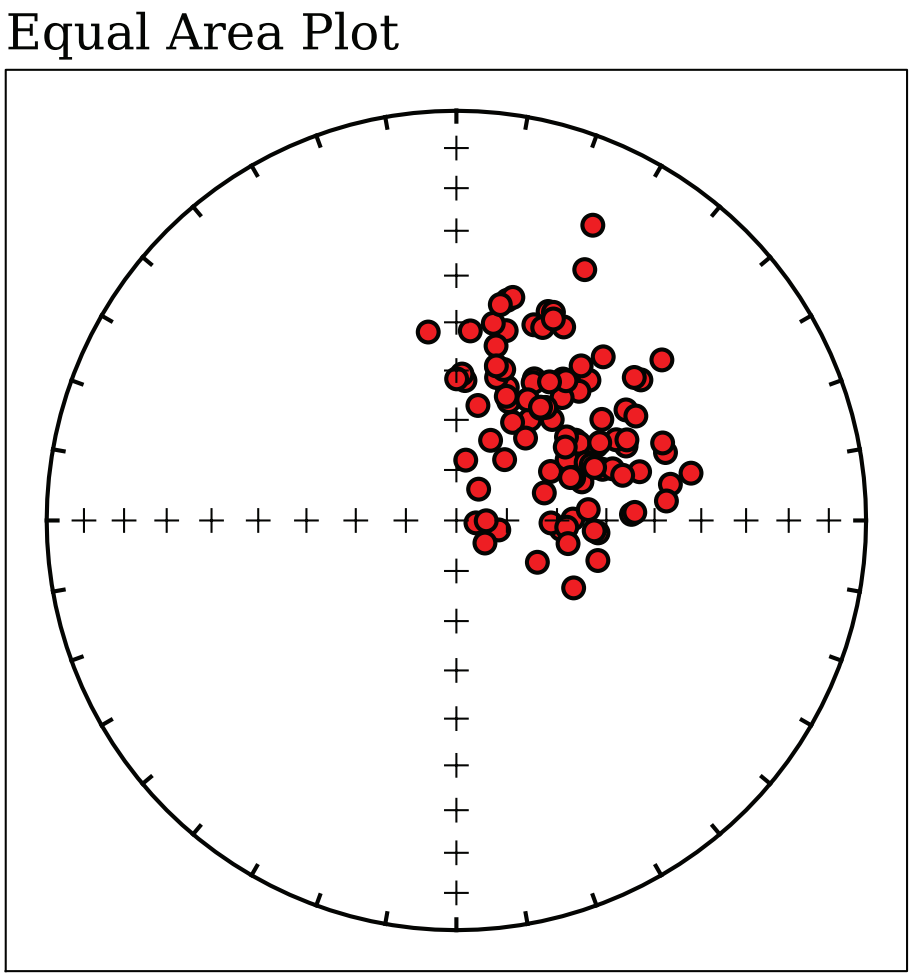
5.2.36 eqarea.py

[Essentials Chapter 2] and [Essentials Appendix B.1] [eqarea docs]

Use the program fishrot.py to generate a Fisher distiributed set of data drawn from a distribution with mean declination of 42° and a mean inclination of 60°. LSave this to a file called it fishrot.out. Use **eqarea.py** to plot an equal area projection of the data.

```
%fishrot.py -D 42 -I 60 > fishrot.out
% eqarea.py -f fishrot.out
```

which produces the plot:



5.2.37 eqarea_ell.py

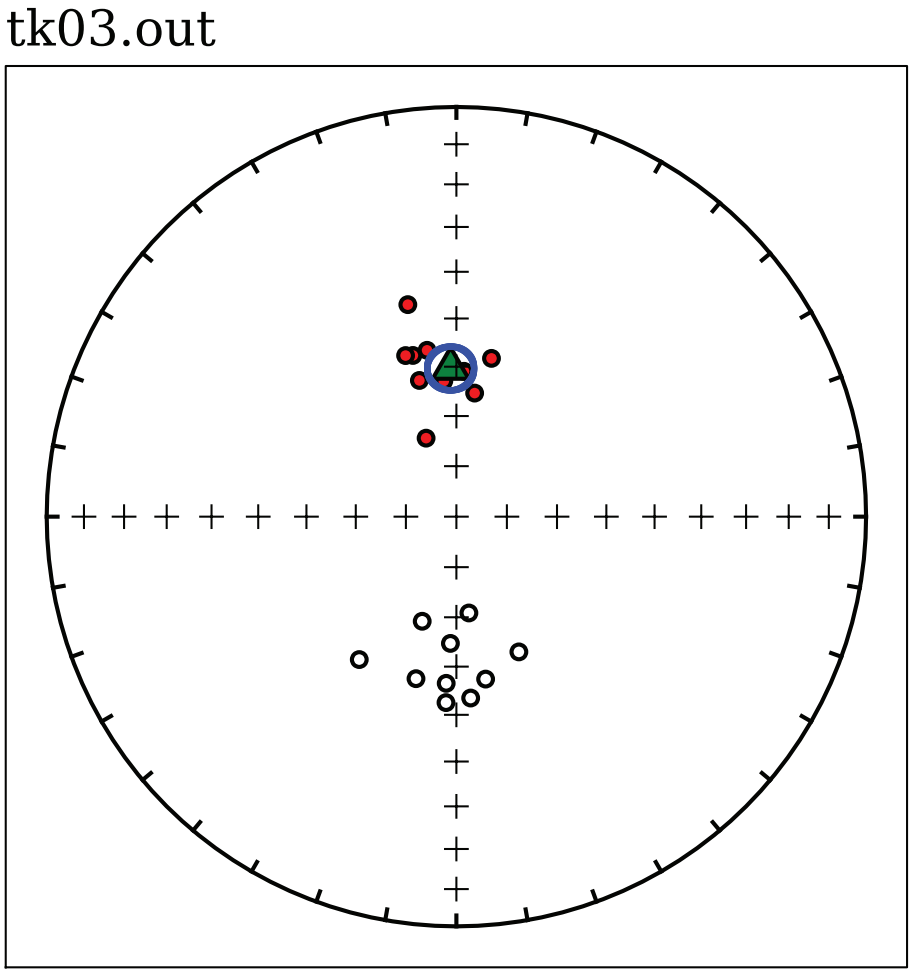
[Essentials Chapters 11] and [Essentials Chapter 12] [eqarea_ell docs]

Use the program tk03.py to generate a set of simulated data for a latitude of 42°N including reversals. Then use the program **eqarea_ell.py** to plot an equal area projection of the directions in *di_example.txt* and plot confidence ellipses. Here is an example for Bingham ellipses.

```
% tk03.py -lat 42 -rev >tk03.out
% eqarea_ell.py -f tk03.out -ell B
     Zdec    21.0
     Edec   105.7
```

```
     Eta     4.5
     n        20
     Einc    10.0
     Zinc   -27.6
     Zeta     4.5
     dec    357.8
     inc     60.3
 S[a]ve to save plot, [q]uit, Return to continue:  a
1  saved in  tk03.out_eq.svg
```

which produces a plot like this:



Other ellipses are Kent, Fisher and bootstrapped ellipses. Check the documentation for details.

5.2.38 eqarea_magic.py

[MagIC] [eqarea_magic docs]

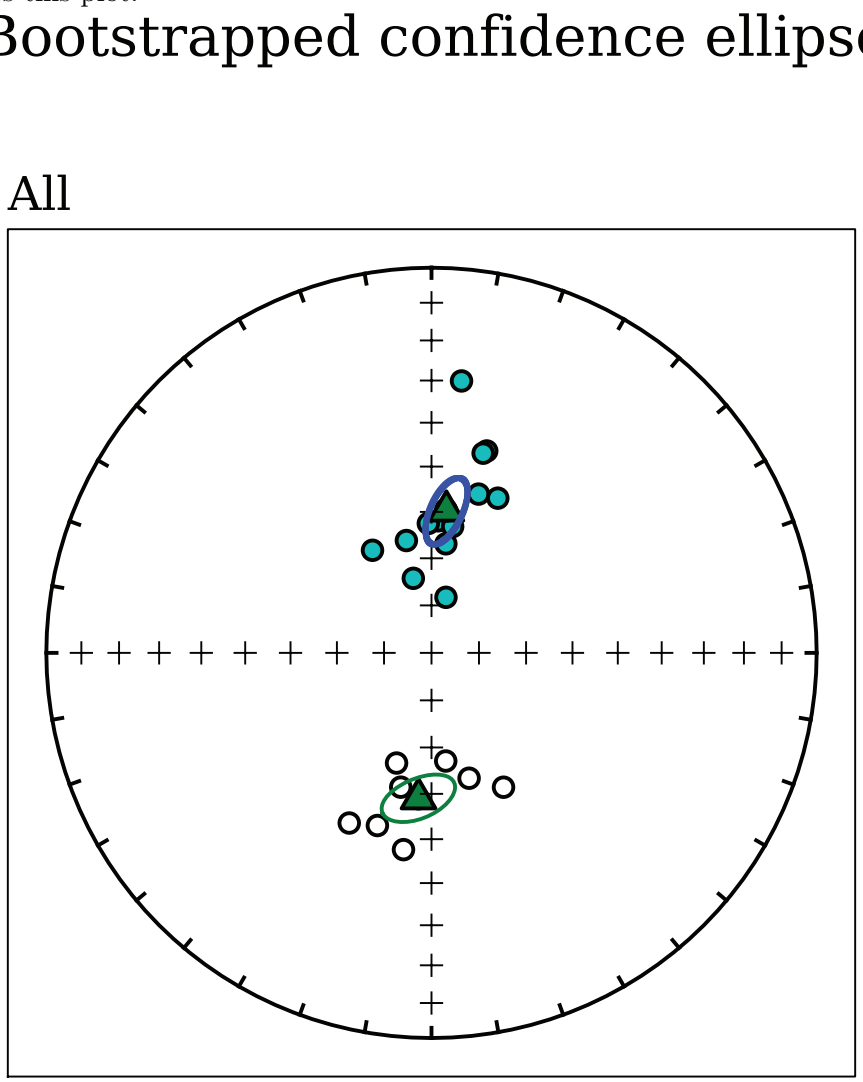
Follow the instructions for downloading and unpacking a data file from the MagIC database or use the file in the *download_magic* directory already

downloaded from the MagIC website. Plot the directional data for the study from the *pmag_results.txt* file along with the bootstrapped confidence ellipse.

```
% eqarea_magic.py -obj loc -crd g -f pmag_results.txt -ell Be
24  records read from  ./pmag_results.txt
All
sr01 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T:LP-PI-ALT-PTRM:LP-PI-TRM-ZI   330.1    64.9
sr03 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T   151.8   -57.5
sr04 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T    16.5    54.6
.
.
.
sr31 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T    23.2    54.0
sr34 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T:LP-PI-ALT-PTRM:LP-PI-TRM-ZI   205.8   -49.4
sr36 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T   197.6   -65.5
sr39 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T:LP-PI-ALT-PTRM:LP-PI-TRM-ZI   188.1   -47.2
sr40 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T   192.9   -60.7
Normal Pole GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T     6.1    59.6
Reverse pole GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T   185.1   -58.8
Grand Mean pole GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T     5.7    59.3
mode  1
     Zdec   111.1
     Edec   206.0
     Eta     7.8
     n         1
     Einc    28.9
     Zinc     8.7
     Zeta     3.3
     dec      6.0
     inc     59.6
mode  2
     Zdec   248.0
     Edec   150.1
     Eta     4.3
     n         1
     Einc    26.4
     Zinc    15.4
     Zeta     8.0
     dec    185.1
     inc    -58.8
```

```
 S[a]ve to save plot, [q]uit, Return to continue:  a
1  saved in  LO:_Snake River_SI:__SA:__SP:__CO:_gu_TY:_eqarea_.svg
```

makes this plot:



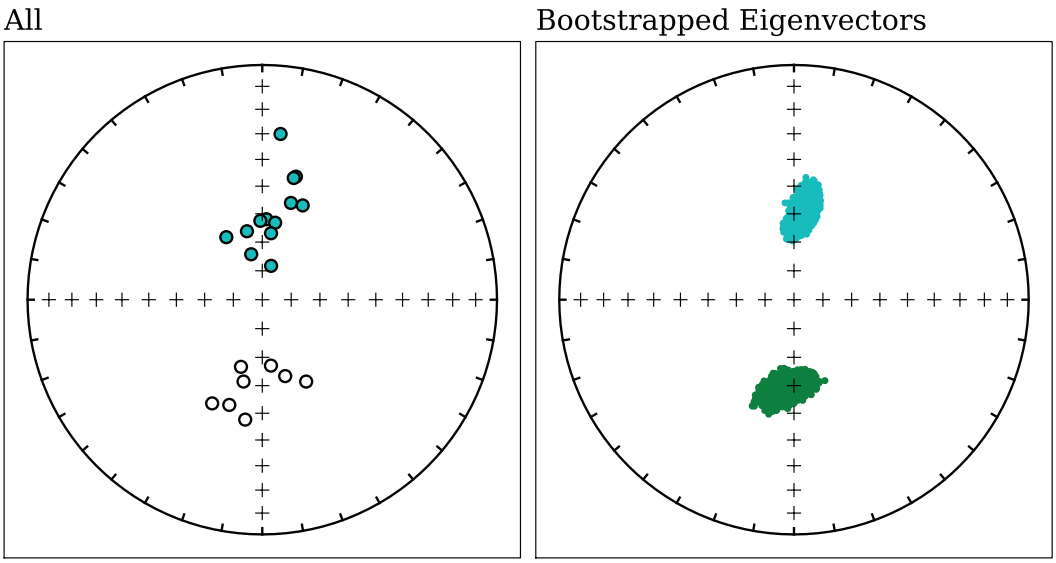
The information printed to the window is the pmag_result_name in the data table, the method codes (here the geochronology method, the type of demagnetization code and the types of demagnetization experiments), and each site mean declination inclination. The information following "mode 1" are the bootstrapped ellipse parameters.

Some data are study averages and some are individual sites.

Use magic_select.py to select only the individual site data. Try the

```
 % magic_select.py -f pmag_results.txt -key data_type i 'T' -F site_results.txt
% eqarea_magic.py -f site_results.txt -ell Bv
21  records read from  ./site_results.txt
All
sr01 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T:LP-PI-ALT-PTRM:LP-PI-TRM-ZI   330.1    64.9
sr03 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T   151.8   -57.5
sr04 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T    16.5    54.6
sr09 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T    14.6    77.9
.
.
.
sr36 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T   197.6   -65.5
sr39 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T:LP-PI-ALT-PTRM:LP-PI-TRM-ZI   188.1   -47.2
sr40 GM-ARAR-AP:LP-DC5:LP-DIR-AF:LP-DIR-T   192.9   -60.7
 S[a]ve to save plot, [q]uit, Return to continue:  a
1  saved in  LO:_Snake River_SI:__SA:__SP:__CO:_g_TY:_eqarea_.svg
2  saved in  LO:_Snake River_SI:__SA:__SP:__CO:_g_TY:_bdirs_.svg
```

which produces plots like:



5.2.39 find_EI.py

[Essentials Chapter 14][find_EI docs]

A data file was prepared using tk03.py to simulate directions at a latitude of 42°. Use the program dipole_pinc.py to find what the expected inclination at this latitude is!

```
% dipole_pinc.py -i
Paleolat for converting to inclination: <cntl-D> to quit 42
   61.0
^D
Paleolat for converting to inclination: <cntl-D> to quit ^D
 Good-bye
```

The data were "flattened" using the formula $\tan I_o = f \tan I_f$ to simulate inclination error and saved in a data file *find_EI_example.dat* in the *find_EI* directory. Use the program **find_EI.py** to find the optimum flattening factor f which, when used to "unflatten" the data yields inclination and elongation (ratio of major and minor eigenvalues of orientation matrix, see the section on eigenvalues in the textbook) most consistent with the TK03.GAD paleosecular variation model of Tauxe and Kent (2004).

```
%  find_EI.py -f find_EI_example.dat
Bootstrapping.... be patient
25  out of  1000
50  out of  1000
75  out of  1000
100  out of  1000
.
.
.
Io Inc  I_lower, I_upper, Elon, E_lower, E_upper
   38.9  =>     58.8 _    48.4 ^    67.0:  1.4679 _ 1.2912 ^ 1.7337
S[a]ve plots - <return> to quit:  a
2  saved in  findEI_ei.svg
3  saved in  findEI_cdf.svg
1  saved in  findEI_eq.svg
4  saved in  findEI_v2.svg
```

which produces these plots:

In this example, the original expected inclination at paleolatitude of 42 (61°) is recovered within the 95% confidence bounds.

5.2.40 fisher.py

[Essentials Chapter 11] [fisher docs]

Draw a set of 10 directions from a Fisher distribution with a κ of 30 using **fisher.py**:

```
% fisher.py -k 30 -n 10
  233.7    81.4
  357.0    76.4
  272.5    62.8
  137.0    70.0
   83.7    71.2
...
```

You could plot the output with, e.g., eqarea.py.

Note that every instance of this program draws a different distribution, so yours will look different in detail.

5.2.41 fishqq.py

[Essentials Chapter 11] [fishqq docs]

Test whether a set of 100 data points generated with fisher.py are in fact Fisher distributed by using a Quantile-Quantile plot:

```
% fisher.py -k 30 -n 100 >fishqq_example.txt
% fishqq.py -f fishqq_example.txt
```

produces these plots:

which support a Fisher distribution for these data.

5.2.42 fishrot.py

[Essentials Chapter 11] [fishrot docs]

Draw a set of 5 directions drawn from a Fisher distribution with a true mean declination of 33, a true mean inclination of 41, and a κ of 50:

```
% fishrot.py -n 5 -D 33 -I 41 -k 50
   35.8    32.8
   36.2    30.2
   37.5    41.8
   28.6    23.9
   26.1    32.8
```

5.2.43 foldtest.py

[Essentials Chapter 12] [foldtest docs]

Use **foldtest.py** to perform the Tauxe and Watson (1994) foldtest on the data in *foldtest_example.dat*.

```
% foldtest.py -f foldtest_example.dat
doing  1000  iterations...please be patient.....
0
50
100
.
.
.

82 - 117 Percent Unfolding
range of all bootstrap samples:  70  -  137
S[a]ve all figures, <Return> to quit   a
3  saved in  fold_ta.svg
2  saved in  fold_st.svg
1  saved in  fold_ge.svg
```

which gives the plots:

Apparently these directions were acquired prior to folding because the 95% confidence bounds on the degree of untilting required for maximizing concentration of the data (maximum in principle eigenvalue) τ_1 of orientation matrix (see the section on eigenvalues in the textbook) includes 100%.

5.2.44 foldtest_magic.py

[Essentials Chapter 12] and [MagIC] [foldtest_magic docs]

This program performs the same test as foldtest.py. The only difference is that it reads MagIC formatted input files and allows the application of selection criteria as specified in the *pmag_criteria.txt* formatted file.

5.2.45 gaussian.py

[Essentials Chapter 11] [gaussian docs]

Use **gaussian.py** to generate a set of 100 normally distributed data points drawn from a population with a mean of 10.0 and standard deviation of 30. Save it to a file named **gauss.out**.

```
% gaussian.py -s 3 -n 100 -m 10. -F gauss.out
```

You can check the sample mean and standard deviation with stats.py or make a histogram of the data with histplot.py

5.2.46 gobing.py

[Essentials Chapter 12] [gobing docs]

Use the dataset generated in the eqarea_ell.py example. Calculate Bingham parameters using **gobing.py** instead of within the plotting program:

```
% gobing.py -f tk03.out
  357.8    60.3     4.5   105.7    10.0     4.5    21.0   -27.6 20
```

which according to the help message from **gobing.py** is:

mean dec, mean inc, Eta, Deta, Ieta, Zeta, Zdec, Zinc, N

5.2.47 gofish.py

[Essentials Chapter 11] [gofish docs]

Draw a set of 10 directions drawn from a Fisher distribution with a true mean declination of 15, a true mean inclination of 41, and a κ of 50 and save it to a file, then use **gofish.py** to calculate the Fisher parameters:

```
%  fishrot.py -n 10 -D 15 -I 41 -k 50 >fishrot.out
% gofish.py -f  fishrot.out
   10.8    39.6    10     9.8484     59.4     6.3    10.5
```

which according to the help message from **gofish.py -h** is: mean dec, mean inc, N, R, k, a95, csd. Your results will vary because every instance of fishrot.py draws a different sample from the Fisher distribution.

5.2.48 gokent.py

[Essentials Chapter 12] [gokent docs]

Draw a set of 20 data points from a TK03.GAD distibution predicted for a latitude of 42°N (see 14), without reversals. Calculate kent parameters using **gokent.py**

```
% tk03.py -n 20 -lat 42 >tk03.out; gokent.py -f  tk03.out
   359.2    55.0     9.3   147.7    30.8     7.8   246.8    14.9 20
```

which according to the help message from **gobing.py** is: mean dec, mean inc, Eta, Deta, Ieta, Zeta, Zdec, Zinc, N

5.2.49 goprinc.py

[Essentials Chapter 12] [goprinc docs]

Draw a set of 20 data points from a TK03.GAD distibution predicted for a latitude of 42°N (see 14), including reversals. Calculate the eigenparameters of the orientation matrix (the principal components) using **goprinc.py**

```
% tk03.py -n 20 -lat 42 -rev >tk03.out; goprinc.py -f tk03.out
0.93863    11.0    58.6 0.04258   226.4    26.5 0.01879   128.3    15.6 20
```

which according to the help message from **gobing.py** is: $\tau_1 V1_D, V1_I, \tau_2 V2_D V2_I \tau_3 V3_D V3_I, N$.

5.2.50 grab_magic_key.py

[MagIC] [grab_magic_key docs]

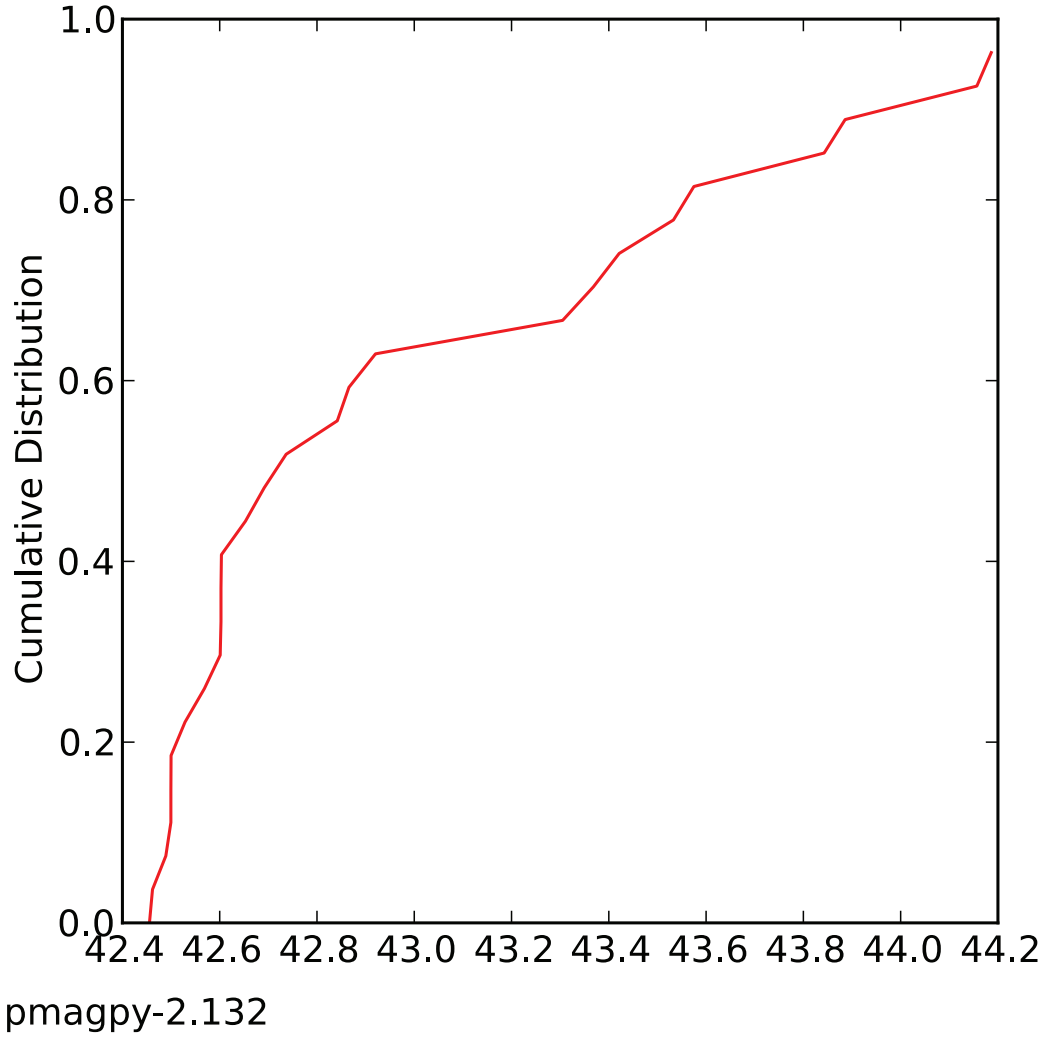
grab_magic_key.py is a utility that will print out any column (-key option) from any [MagIC] formatted file. For example, we could print out all the site latitudes from the *er_sites.txt* file down loaded in the download_magic.py example:

```
% grab_magic_key.py -f er_sites.txt -key site_lat
42.60264
42.60264
42.60352
42.60104
...
```

You could save the data in a file with the output redirect function (>) and plot them with, say plot_cdf.py.

```
% grab_magic_key.py -f er_sites.txt -key site_lat > lats
% plot_cdf.py -f lats
```

which produces the fascinating (NOT!) plot:



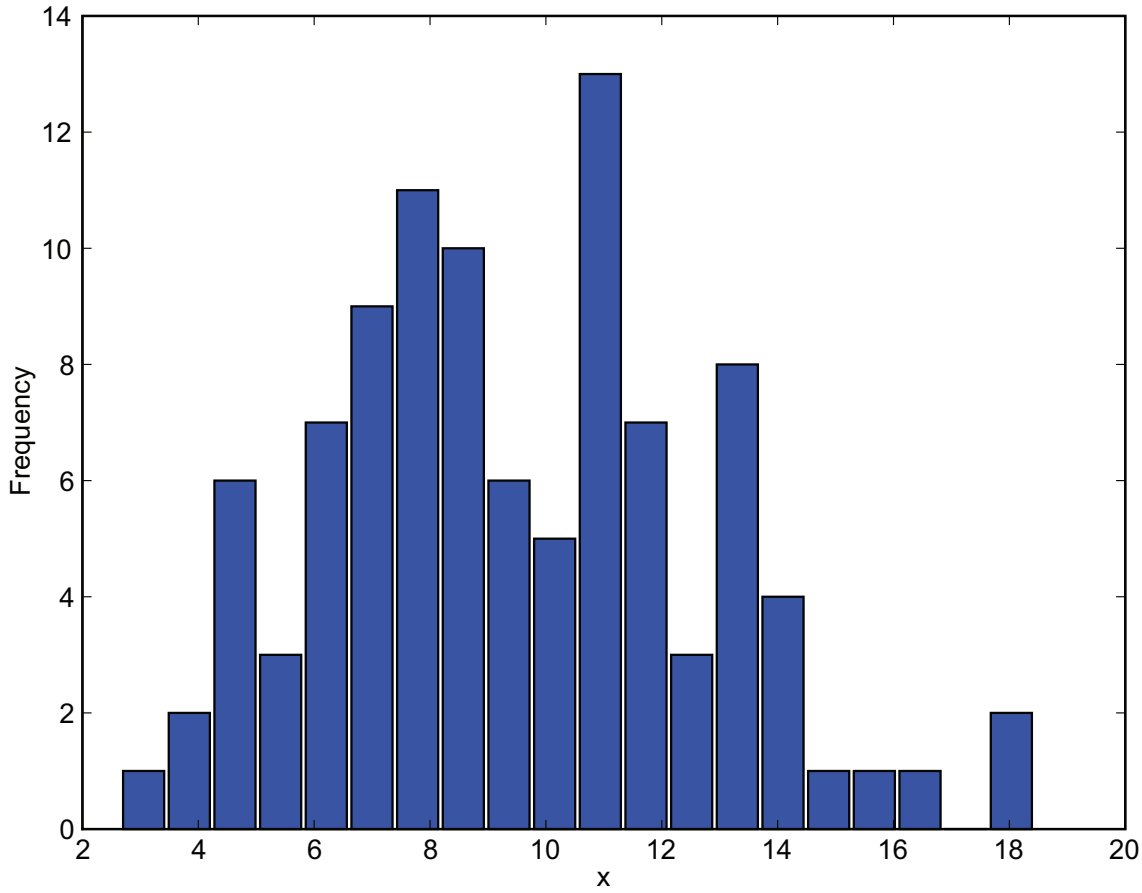
5.2.51 histplot.py

[histplot docs]

Make a histogram of the data generated with the **gaussian.py** program.

```
% histplot.py -f gauss.out
```

which makes a plot similar to:



5.2.52 hysteresis_magic.py

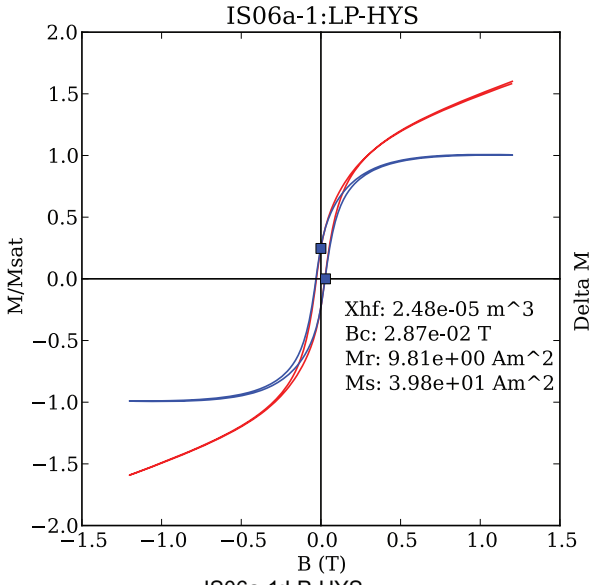
[Essentials Chapters 5], [Essentials Chapter 7], [Essentials Appendix C.1] & [MagIC] [hysteresis_magic docs]

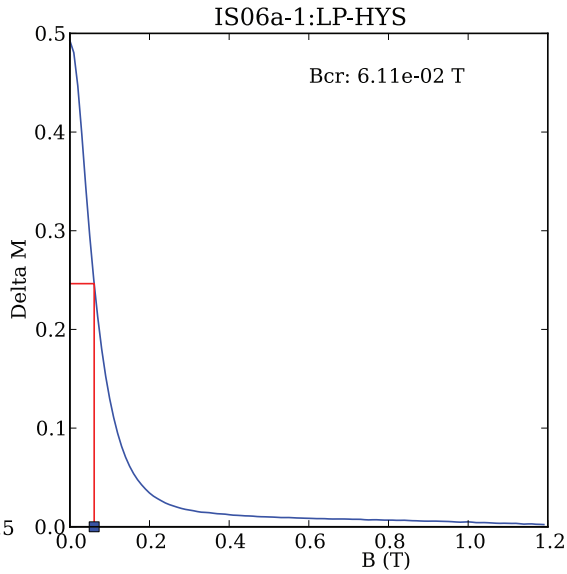
Plot the data in *hysteresis_magic_example.txt*, (from Ben-Yosef et al., 2008) which were imported into MagIC using AGM_magic.py. Use the program **hysteresis_magic.py** to plot the data.

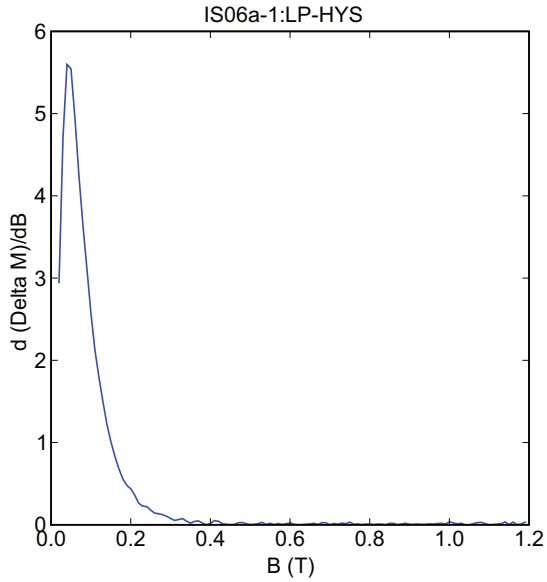
```
% hysteresis_magic.py -f hysteresis_magic_example.dat

IS06a-1 1 out of  8
S[a]ve plots, [s]pecimen name, [q]uit, <return> to continue
 a
1  saved in  _IS06a-1_hyst.svg
```

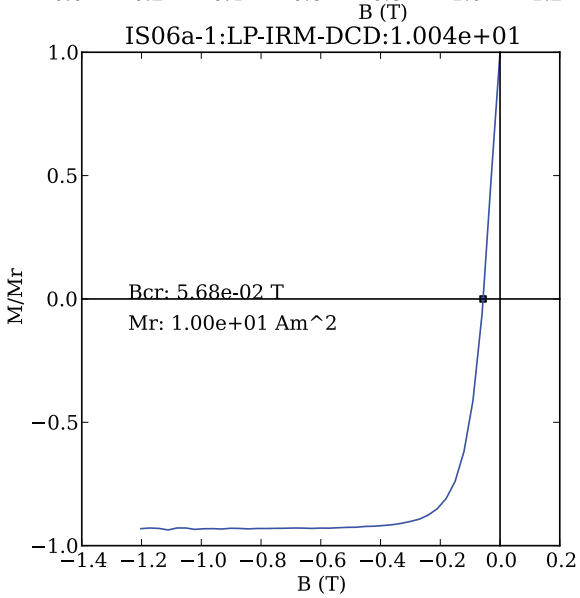
```
3  saved in  _IS06a-1_DdeltaM.svg
2  saved in  _IS06a-1_deltaM.svg
5  saved in  _IS06a-1_irm.svg
```

which makes the plots:









5.2.53 igrf.py

[Essentials Chapter 2][Essentials Chapter 2] [igrf docs]

Use the program **igrf.py** to estimate the field on June 1, 1995 in Amsterdam, The Netherlands (52.5° N, 5°E).

```
% igrf.py -i
Decimal year: <cntrl-D to quit> 1995.5
Elevation in km [0]
Latitude (positive north) 52.5
Longitude (positive east) 5
  357.9    67.4    48534

Decimal year: <cntrl-D to quit> ^D
 Good-bye
```

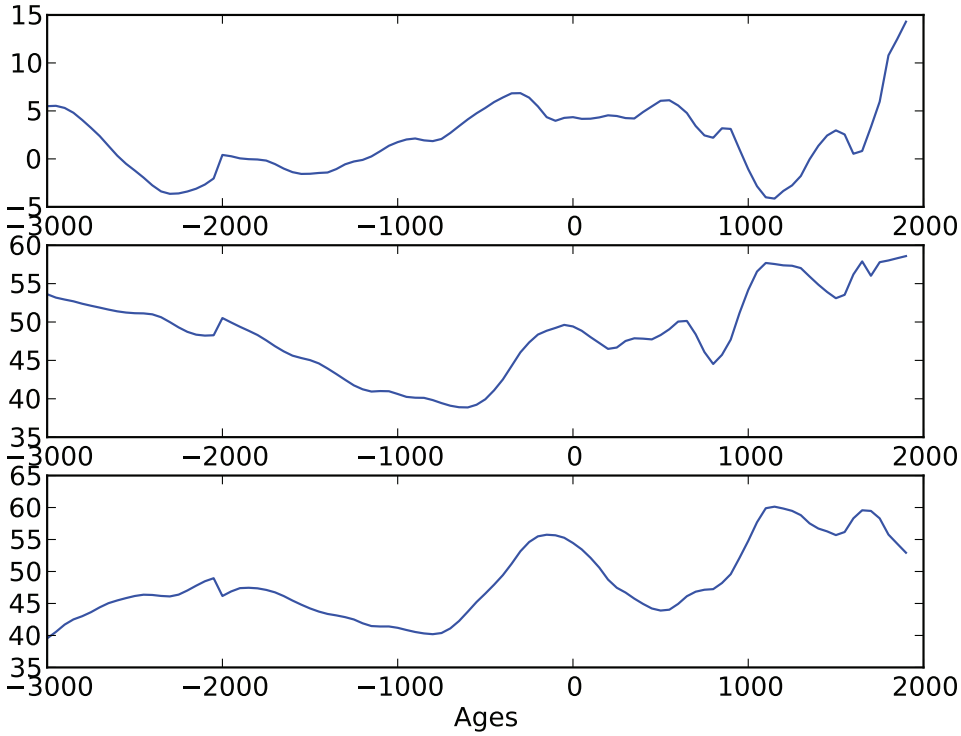
Now read from the file *igrf_example.dat* which requests field information for near San Diego for the past 3000 years. The program will evaluate the field vector for that location for dates from 1900 to the present using the IGRF/DGRF coefficients in the IGRF-11 model available from: `http://www.ngdc.noaa.gov/IAGA/vmod/igrf.html`. For the period from 1000 BCE to 1800, the program uses the CALS3k.4b of Korte and Constable (2011) available at: `http://earthref.org/ERDA/1142/`. Prior to that and back to 8000 BCE, the program uses the coefficients of Korte et al., 2011 for the CALS10k.1b, available at: `http://earthref.org/ERDA/1403/`

```
% igrf.py -f igrf_example.dat -F igrf.out
```

You can also make a plot of secular variation of field elements by specifying the age range and location on the command line. Try this for the location of San Diego for the age range: -3000 1925 in increments of 50 years:

```
% igrf.py -ages -3000 1950 50 -loc 33 -117 -plt
     S[a]ve to save figure, <Return>  to quit
```

and get this plot:



5.2.54 incfish.py

[Essentials Chapter 11] [incfish docs]

Use the program **incfish.py** to calculate average inclination for inclination only data simulated by fishrot.py for an average inclination of 60°. If you save the declination, inclination pairs, you can compare the **incfish.py** answer with the Fisher mean. The datafile *incfish_example_di.dat* has the declination, inclination pairs and *incfish_example_inc.dat* has just the inclinations.

```
% fishrot.py -I 60 -k 10 > incfish_example_di.dat
% awk '{print $2}' incfish_example_di.dat > incfish_example_inc.dat
% incfish.py -f incfish_example_inc.dat
   57.1    61.0  100     92.9    13.9     1.0
% gofish.py -f incfish_example_di.dat
    1.8    62.9    100    91.2727     11.3     4.4    24.0
```

The output for **incfish.py** is: [gaussian] mean inc, Fisher inc, N, R, k, α_{95}. You can see that the **incfish.py** result is much closer to the Fisherian result than the gaussian mean is.

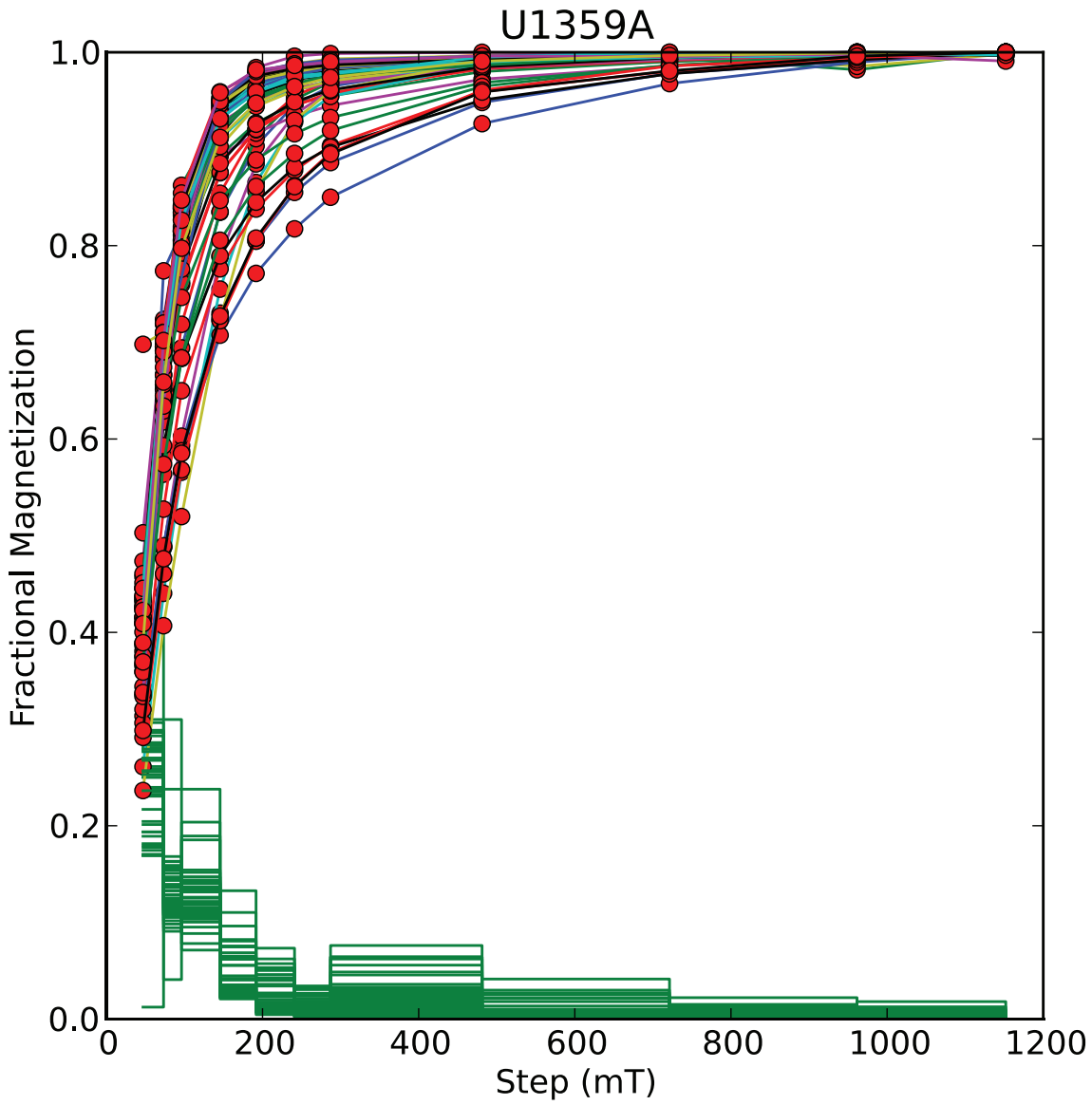
5.2.55 irmaq_magic.py

MagIC [irmaq_magic docs]

Someone (Saiko Sugisaki) measured a number of samples from IODP Expedition 318 Hole U1359A for IRM acquisition curves. She used the ASC impulse magnetizer's coils # 2 and 3 and saved the data in the SIO format. Convert these to the MagIC measurements format, combine them in a single *magic_measurements.txt* file with combine_magic.py and plot the data with **irmaq_magic.py**.

```
% sio_magic.py -f U1359A_IRM_coil2.txt -LP I -V 2 -F U1359A_IRM_coil2.magic \
   -loc U1359A -spc 0 -ncn 5
% sio_magic.py -f U1359A_IRM_coil3.txt -LP I -V 3 -F U1359A_IRM_coil3.magic \
    -loc U1359A -spc 0 -ncn 5
% combine_magic.py -F magic_measurements.txt -f U1359A_IRM_coil2.magic \
    U1359A_IRM_coil3.magic
% irmaq_magic.py
559  records read from  magic_measurements.txt
U1359A
 S[a]ve to save plot, [q]uit,  Return to continue:  a
1  saved in  U1359A_LP-IRM.svg
```

which produces the plot:



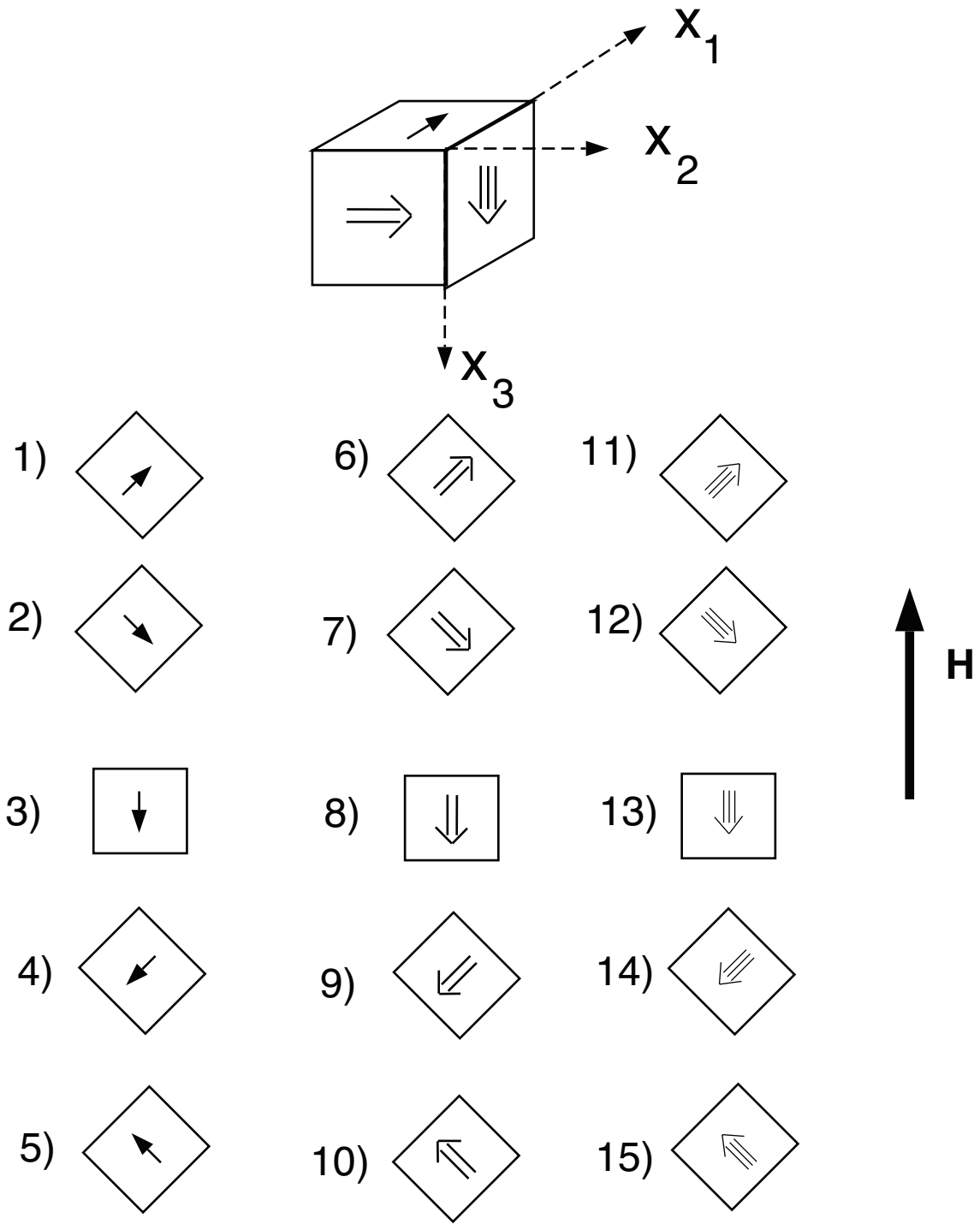
5.2.56 k15_magic.py

[Essentials Chapter 13]; [MagIC] [k15_magic docs]

Someone took a set of samples from a dike margin in the Troodos Ophiolite and measured their anisotropy of magnetic susceptibility on an a Kappabridge KLY 2.0 instrument in the SIO laboratory:

```
tr245f     80.00   -46.0 204 25
   995.     999.     993.     995.    1000.
  1004.     999.    1001.    1004.     999.
   998.     997.    1002.     998.     997.
tr245g     52.00   -23.0 204 25
  1076.    1066.    1072.    1077.    1067.
  1076.    1068.    1072.    1076.    1067.
  1068.    1075.    1071.    1068.    1076.
tr245h    123.00   -29.0 204 25
  1219.    1231.    1218.    1220.    1232.
  1230.    1231.    1234.    1230.    1231.
  1221.    1221.    1225.    1221.    1221.
tr245i1    68.00   -26.0 204 25
  1031.    1027.    1025.    1032.    1028.
  1035.    1031.    1035.    1035.    1032.
  1030.    1027.    1032.    1030.    1027.
```

The first line of each set of four has the specimen name, azimuth, plunge, and bedding strike and dip the next three lines are sets of five measurements in the 15 positions recommended by Jelinek (1977):


X1
X2
X3
1)
2)
3)
4)
5)
6)
7)
8)
9)
10)
11)
12)
13)
14)
15)
H

The 15 measurements for each specimen, along with orientation information and the specimen name were saved in the file *k15_example.dat*. Convert these to the MagIC format using the program **k15_magic.py**:

```
% k15_magic.py -spc 0 -f k15_example.dat -loc "Troodos Ophiolite"
Data saved to:  ./er_samples.txt ./rmag_anisotropy.txt ./rmag_results.txt ./k15_measureme
```

You can plot the output of this example (default file *rmag_anisotropy.txt*) using the program **aniso_magic.py**.

5.2.57 k15_s.py

[Essentials Chapter 13]; [MagIC] [k15_s docs]

Use **k15_s.py** to calculate the best-fit tensor elements and residual error for the data in the file *k15_example.dat* (same file as for k15_magic.py. These are: the specimen name, azimuth and plunge and the strike and dip, followed by the 15 measurements made using the Jelinek 1977 scheme shown in the k15_magic.py example. Calculate the .s data in specimen, geographic and tilt adjusted coordinates:

```
% k15_s.py -f k15_example.dat
0.33146986 0.33413991 0.33439023 0.00075095 -0.00083439 -0.00016688 0.00008618
0.33335925 0.33335925 0.33328149 -0.00155521 -0.00132193 0.00116641 0.00017193
0.33097634 0.33573565 0.33328801 0.00163177 0.00013598 0.00000000 0.00018131
...
% k15_s.py -f k15_example.dat -crd g
0.33412680 0.33282733 0.33304587 -0.00015289 0.00124840 0.00135721 0.00008618
0.33556300 0.33198264 0.33245432 0.00087259 0.00024141 0.00096166 0.00017193
0.33584908 0.33140627 0.33274469 0.00131844 0.00118816 0.00002987 0.00018131
...
% k15_s.py -f k15_example.dat -crd t
0.33455712 0.33192658 0.33351630 -0.00043563 0.00092770 0.00105006 0.00008618
0.33585501 0.33191562 0.33222935 0.00055960 -0.00005314 0.00064730 0.00017193
0.33586666 0.33084926 0.33328408 0.00142269 0.00013235 0.00009201 0.00018131
...
```

5.2.58 KLY4S_magic.py

[Essentials Chapter 13]; [MagIC] [KLY4S_magic docs]

The program **AMSSpin** available for downloading from `http://earthref.org/ERDA/940/` generates data for the Kappabridge KLY4S spinning magnetic susceptibility instrument as described by Gee et al. (2008).

The files have the format:

```
318-U1359B-011H-1-W-75  0.3415264 0.3403534 0.3181202 -0.0006608 0.0000948 0.0023
318-U1359B-011H-2-W-76  0.3369955 0.3361130 0.3268916 0.0000845 -0.0005236 0.0003
318-U1359B-011H-3-W-78  0.3373485 0.3364144 0.3262371 -0.0002281 0.0005881 -0.001
318-U1359B-011H-4-W-96  0.3343428 0.3323376 0.3333195 0.0007210 -0.0005503 0.0011
```

where the columns are:

Specimen S_1 S_2 S_3 S_4 S_5 S_6 $\chi_b(\mu$SI) date time user

Output files are in the format of the file *KLY4S_magic_example.dat* (found in the Measurement_Import/KLY4S_magic folder). One option is for orientation information to be output as an *azdip* formatted file (see azdip_magic.py.) Try it out on some data from IODP Expedition 318 (Hole U1359B) published by Tauxe et al., (2012). Note that the sample naming convention is # 5 (-ncn 5) and the sample name is the same as the site (-spc 0) using **KLY4S_magic.py** as follows:

```
%KLY4S_magic.py -f KLY4S_magic_example.dat  -spc 0 -ncn 5
```

This command will create the files needed by the MagIC database and the data can be plotted using **aniso_magic.py**. If you were to import the sample files from the LIMS data base for these samples, you could plot them versus depth, or as equal area projections using ani_depthplot.py and aniso_magic.py respectively.

5.2.59 lnp_magic.py

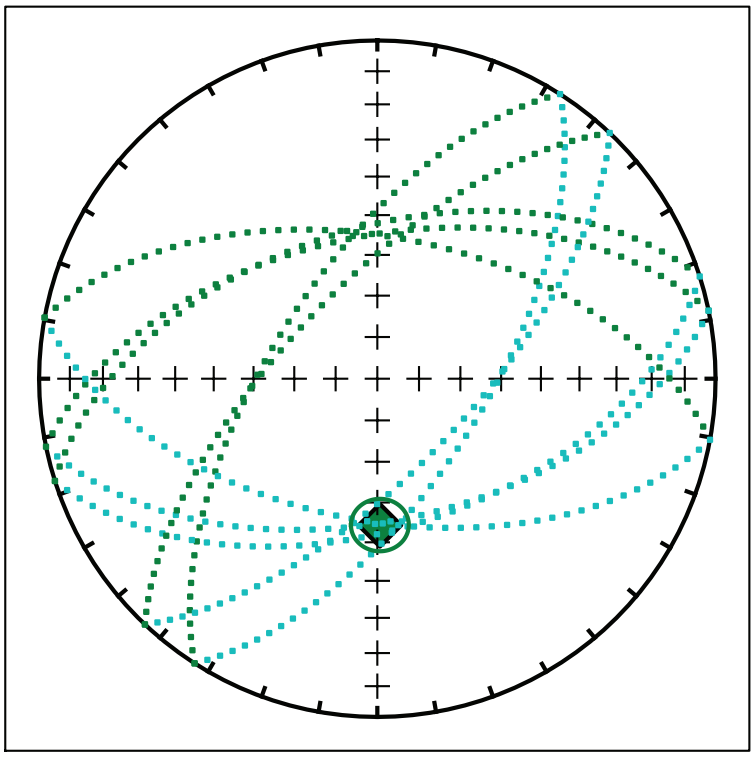
[Essentials Chapter 11], [Essentials Appendix C2.2] & [MagIC] [lnp_magic docs]

This program will take *pmag_specimen* formatted MagIC files (for example, generated by zeq_magic.py) and plot data by site, combining best-fit lines and best-fit planes using the method described in Essentials Appendix C2.2. Try this out on the data from the San Francisco Volcanics, notorious for lightning strikes, published by Tauxe et al., 2003. These can be downloaded from the MagIC database from `http://earthref.org/MAGIC/m000629dt20061213090720/` and unpacked with download_magic.py.

```
% download_magic.py -f zmab0001193tmp02.txt
working on:  'er_locations'
er_locations  data put in  ./er_locations.txt
working on:  'er_sites'
er_sites  data put in  ./er_sites.txt
working on:  'er_samples'
er_samples  data put in  ./er_samples.txt
...
% lnp_magic.py -f pmag_specimens.txt -crd g
sv01
Site lines planes  kappa   a95   dec   inc
sv01 0  5     286      6.6    179.0    -54.3  4.9948
% tilt correction:  0
s[a]ve plot, [q]uit, <return> to continue:
 a
1  saved in  sv01_g_eqarea.svg
```

which generated this figure:

sv01: geographic coordinates



5.2.60 lowrie.py

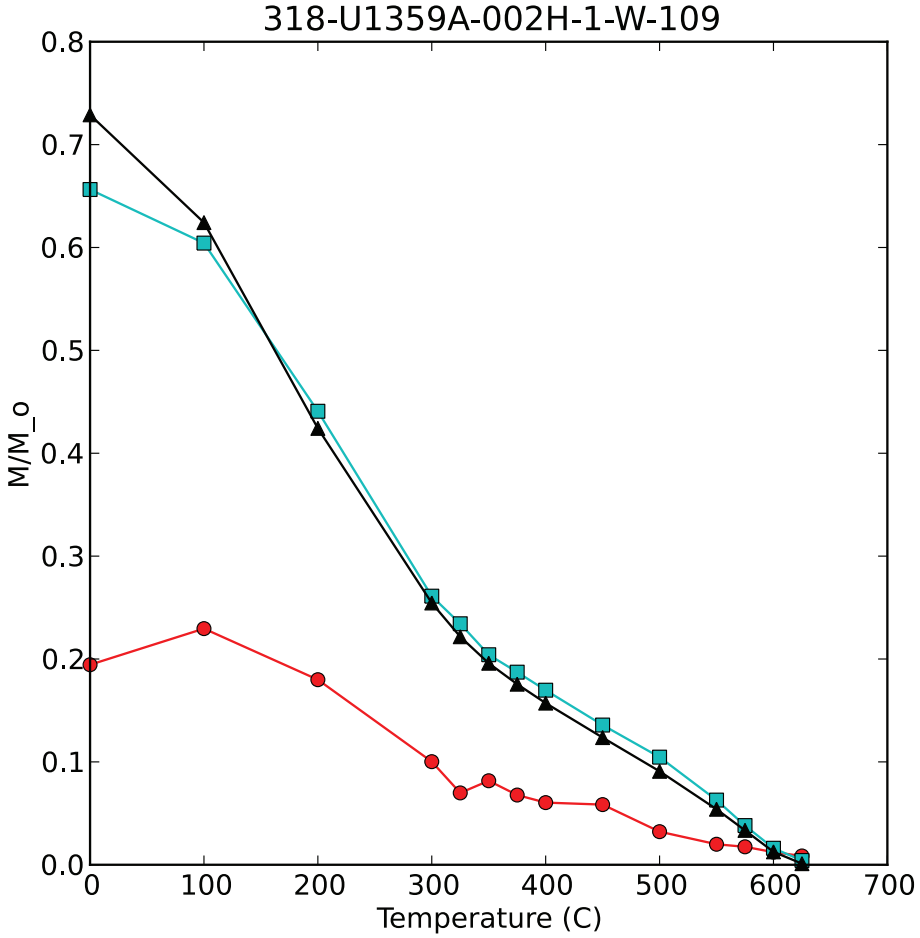
[Essentials Chapter 8] [lowrie docs]

Someone (Saiko Sugisaki) subjected a number of specimens from IODP Expedition 318 Hole U1359A specimens to a 3-D IRM experiment and saved the data in the SIO format. Use **lowrie.py** to make plots of blocking temperature for the three coercivity fractions.

```
% lowrie.py -f lowrie_example.dat
318-U1359A-002H-1-W-109
S[a]ve figure? [q]uit, <return> to continue   a
1  saved in  lowrie:_318-U1359A-002H-1-W-109_.svg
```

```
318-U1359A-002H-4-W-65
S[a]ve figure? [q]uit, <return> to continue   q
```

which produces plots like this:



5.2.61 lowrie_magic.py

[Essentials Chapter 8] and [MagIC] [lowrie_magic docs]

This program works exactly like lowrie.py, but works on *magic_measurements.txt* formatted files. Use sio_magic.py to import the *lowrie_example.dat* data file into the MagIC format. Then use **lowrie_magic.py** to plot the data:

```
% sio_magic.py -f lowrie_example.dat -LP I3d -loc U1359A -F lowrie_magic_example.dat
averaging: records  11 12
```

```
318-U1359A-004H-2-W-18 started with  15  ending with  14
Averaged replicate measurements
averaging: records  13 14
318-U1359A-004H-2-W-44 started with  15  ending with  14
Averaged replicate measurements
averaging: records  14 16
318-U1359A-012H-5-W-70 started with  16  ending with  14
Averaged replicate measurements
averaging: records  14 15
% lowrie_magic.py -f lowrie_magic_example.dat
318-U1359A-002H-1-W-109
S[a]ve figure? [q]uit, <return> to continue
```

5.2.62 MagIC.py Tutorial

MagIC MagIC.py

Importing data into the MagIC Project Directory

Data do not magically appear in the database. The paleomagnetic community must contribute them. One way to accomplish this is to use PmagPy to import measurement data into the MagIC format. You can use the MagIC.py GUI for this purpose. Find some example datafiles in the subdirectory 'PmagPy_tutorial' of the Datafiles_2.0 directory. Within this subdirectory, you will find a directory called '*MyFiles*'. This contains data from alternating field demagnetization (*mc_af.mag*), thellier-type experiments (*mc_thel.mag*), anisotropy of anhysteretic remanence (*mc.aarm*), hysteresis loops (*mc205a1-1.agm* and *mc217a2-1.agm*) as well as a file containing field orientation, location, etc. (*orient.txt*), age information (*er_ages.txt*) and citation information (*er_citations.txt*).

Follow the steps outlined in Quick Start to create a Project Directory and launch the **MagIC.py** GUI. Create a directory called *MagIC* in your Project Directory. Different operating systems will have a different look, but all versions will put up a Welcome window when you have fired up the program MagIC.py.

Now import the following:

- Field and sampling information: orient.txt.
  - Select orientation file: choose your orient.txt file from MyFiles.

- Select naming convention: In this tutorial choose the first naming convention XXXXY (e.g., sample mc200a is related to site mc200 by a terminal character).

- Select orientation convention...: Choose to use the IGRF calculation at the site and enter 13 hours for the hours to add to get from McMurdo time to GMT. These samples were oriented with a Pomeroy orientation device so the correct orientation method is #1, the default.

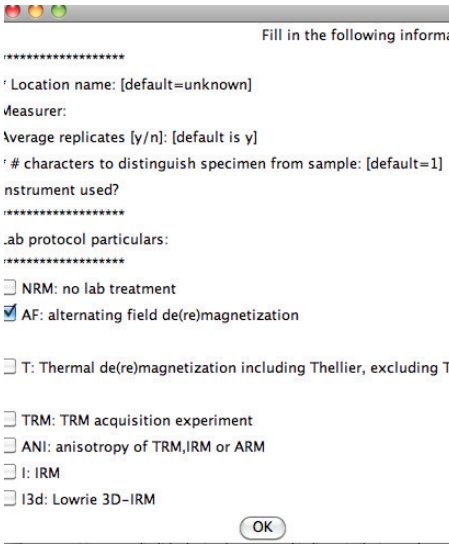
- Select method codes: Check 'FS-FD' and 'SO-POM'. All the site locations were determined with a GPS.

- Importing magnetometer data: For the example here, we will use the SIO format and import the following files: *mc_af.mag, mc_thel.mag, mc_aarm.mag* which are the AF demagnetization data, Thellier experiments, and anisotropy of anhysteretic magnetization respectively.

  - Select naming convention: Choose the first option (XXXXY).

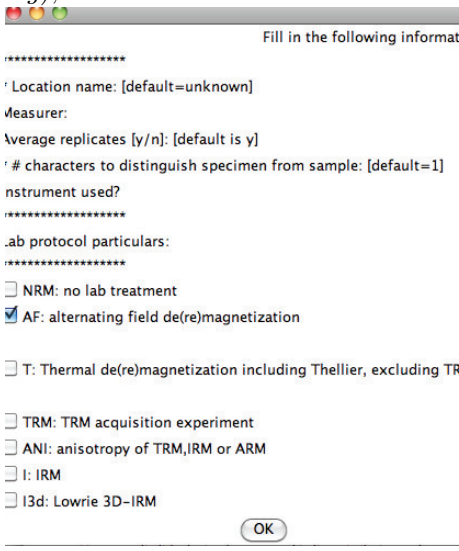
  - Next you will be presented with forms a form to file out with details like the location name (here, McMurdo) and laboratory protocol particulars. It is strongly encouraged to fill in the location name as this will save much head ache later on. In the example files provided here, specimen names are all related to the sample names by a terminal single character (specimen mc200a2 came from sample mc200a), so the number of characters to distinguish a specimen from a sample is '1'. This is the default value so you don't actually have to enter the '1' in the forms for this example.

  - Check the appropriate protocols for each file.



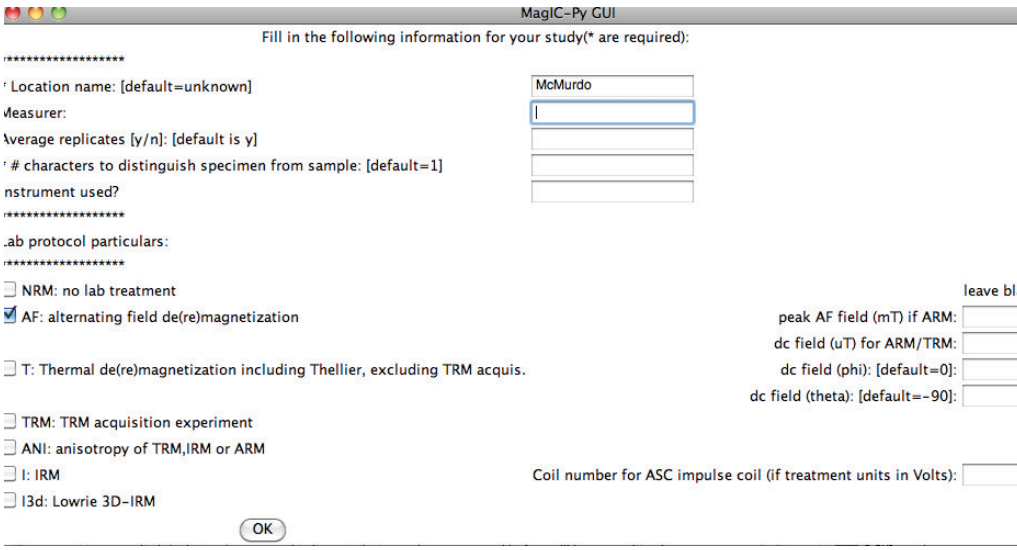
* For the AF file (*mc_af.mag*), check the AF box:

* For the Thellier experimental data (*mc_thel.mag*), check the



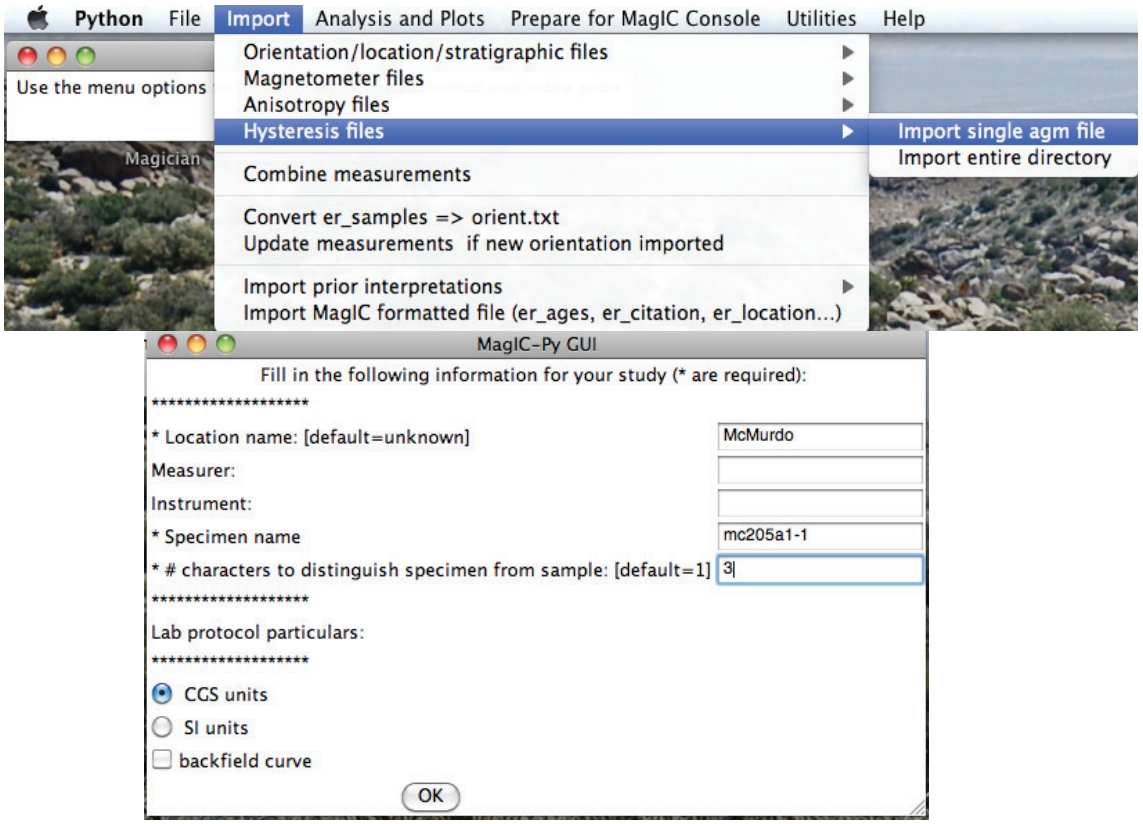
'T' box and fill in the DC field value (30 muT):

* For the AARM data, check the 'ANI' box and the 'AF' box and fill in the peak AF (180 mT) and dc field values (50 μT)



to the right:

- Importing hysteresis data There are two different modes for importing hysteresis/backfield data: by file or by directory. As there are only two in this example, we will use the former:



  - Select the file.
  - Select the naming convention. Choose the first option as before.
  - Enter the location name (McMurdo), and the specimen name (same as file name) and number of characters to delineate the specimen from the sample (3, in this example).
  - Indicate that these files are in cgs units).

- To combine all the data together, select Combining measurement files.
- Import Prior Interpretations: There are several ways to import prior interpretations in MagIC.py. One of the ways is to use the output of thezeq_magic.py and thellier_magic.py programs which interpret directional (AF and thermal) and paleointensity (Thellier-type) experimental data respectively. They produce *pmag_specimen* formatted files called *zeq_specimens.txt* and *thellier_specimens.txt* respectively. The original interpretations from the Lawrence et al. (2009) paper are in

the MyFiles directory. These can be imported by choosing the 'Import prior interpretations' menu and choosing the MagIC formatted specimen file option.

- Importing MagIC formatted files: There are two other MagIC formatted files already filled out for this study: *er_citations.txt* and *er_ages.txt*. The first has the relevant citation information, and the second age determinations for particular sites Import these files into the MagIC directory using the 'Import MagIC formatted file' option.

After you have imported your various datafiles, you can process them using the tools in the Analysis and Plots menu. All of these options generate and execute command line calls for various PmagPy programs. The GUI uses default options because it "knows" the names of the files it has created, saving you from having to figure that out. Many of these options require you to type things in on the command line, such as 'a' to s[a]ve plots, 'q' to quit, or simply return to increment the specimen name. Follow the instructions for the specific program that are printed in the terminal window (see also the documentation in the section on PmagPy programs). Once the program has exited, control will be returned to the MagIC.py GUI.

The plots that are available to you now are: Demagnetization data, Thellier-type experimental data, and Hysteresis data. You may change the selection criteria from the defaults if you wish.

When you are finished looking at and perhaps re-interpreting the data, you are ready to prepare the data for the MagIC Console: the gateway to the MagIC database. This is a three step process under the Prepare for MagIC Console menu: : Assemble specimens, Assemble results and Prepare upload file.

- Assemble specimens: Just choose this and watch the console for errors.
- Assemble results: Fill in the following:
  - Choose the default selection criteria (unless you changed them, in which case you can select the customized option -exc).
  - Age Range: These samples ranged from zero to five million years, so fill in '0', '5' and choose 'Ma' under the
  - Choose the present latitude (-lat option) for calculating VADMs.
  - Coordinate system: Choose the geographic coordinate system.
  - If you want to look at the data on a site by site basis, select the -p option.

- Prepare upload txt file: The MagIC Console will import a specially formatted ascii file, parsing all the tables into their correct tabs and allowing you to proceed to the final steps of file preparation for uploading into the database. It will create an *er_locations.txt* file for you based on what is in the *er_sites.txt* file. It will ask you for the missing required information. In this case, the location type was 'Outcrop'. Then the GUI prepares a file called *upload_dos.txt* which can be directly imported into the MagIC console.

After you have assembled the results some additional equal area projections and utilities will be available to you, if you wish to try these out.

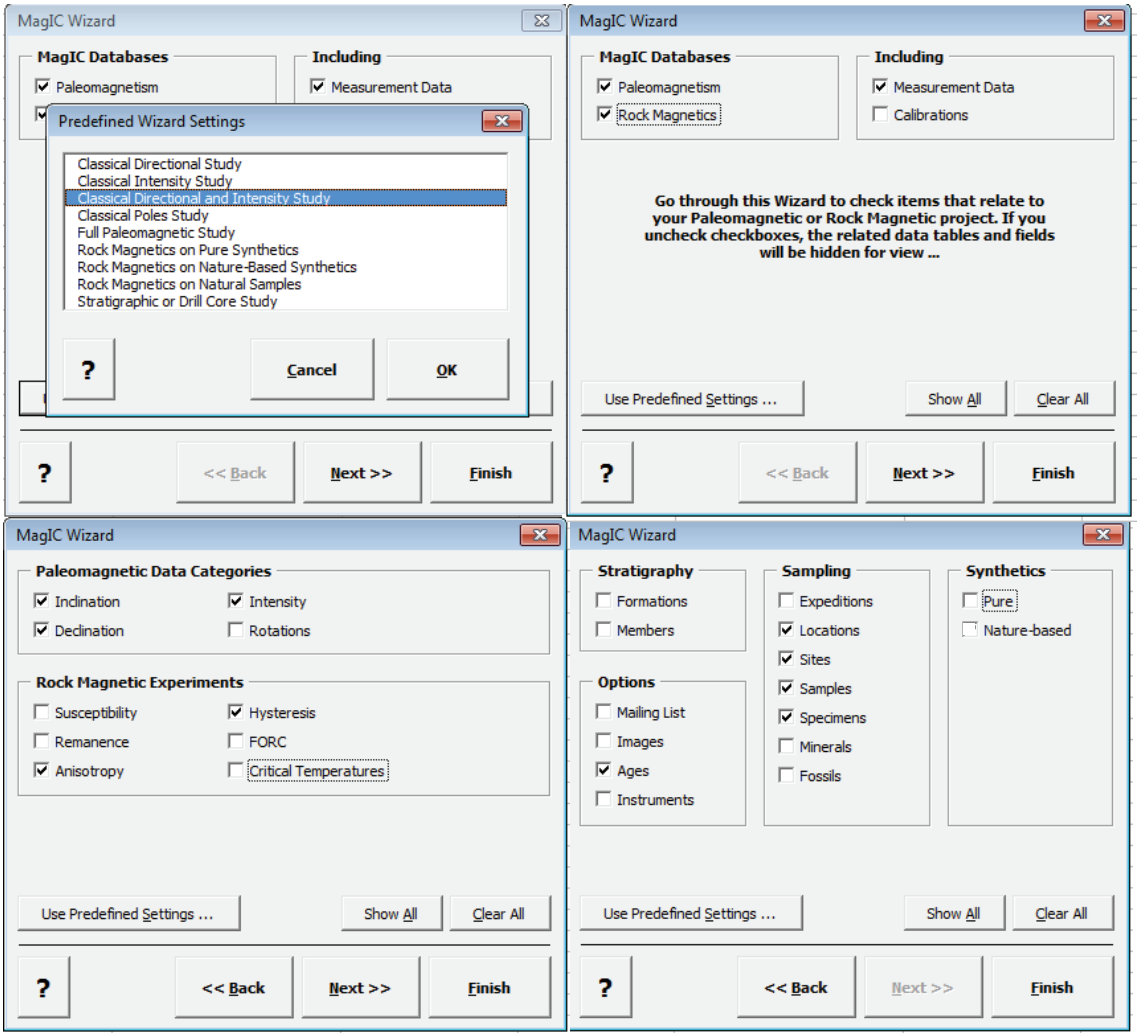
Using the MagIC Console

To obtain the MagIC Console software, go to:

`http://earthref.org/MAGIC`

and select 'MagIC Software v2.4'. Click on the 'CONSOLE SOFTWARE' and 'CONTROLLED VOCABULARIES' links. Unzip the Console software and put the Controlled vocabularies file into it (copying over whatever was there). As of this writing the Console works on both Macs and PCs, but is extremely slow on the Mac OS operating system. Having been warned, open Excel and then select the 'MagIC.v24.console' file. This will open the console. Under 'Operations', select 'new smartbook' and type the name 'Tutorial' in the File name text box.

The 'wizard' will guide you in choosing the necessary columns for the smartbook:



Click on the 'Use predefined settings...' choose 'Classical Directional and Intensity Study' and click 'OK'. Then add 'Measurement data' and 'Rock Magnetics' and click on the 'Next>>' button. Check the boxes for 'Anisotropy' and 'Hysteresis' data and click on the 'Finish' button. Your smartbook is now ready for populating.

Under the 'Operations' menu, select 'Import data files' and choose the 'upload_dos.txt' file in your MagIC project directory. This reads it in and parses it out to the right tables. Any error logs generated will be in the directory with the upload_dos file. When you have satisfied your curiosity about the contents of the smartbook, click on the 'Prepare for uploading' option under the 'Operations' menu. After you indicate that you really want

to do this, the console goes through some extensive checking. If errors are encountered, you will be given the opportunity to correct them. When the console is finished, it exports an Excel file (in this case Tutorial.v24.xls) and a text file (in this case Tutorial.v24.txt). If this were a real study, you would upload these into the MagIC database by going to: `http://earthref.org/MAGIC` and selecting 'Upload a new contribution'. Do NOT upload this file to the database, as all uploaded datasets must be associated with publications. This was just a Tutorial.

5.2.63 magic_select.py

[MagIC] [magic_select docs]

This program takes any MagIC formatted file and selects lines that match the search criteria and saves the output as a new MagIC formatted file of the same type. Selection criteria match whole or partial strings, avoid whole or partial strings or evaluate minimum or maximum values. The syntax is to specify the input file with the -f option, the key you wish to search on with -key, followed by the string you wish to use as a criterion, followed with a 'T' for match exactly, 'F' avoid entirely, 'has' for matching partially and 'not for avoiding any partial match, and min, max and eval for evaluating the numerical value. The -F option sets the output file.

For example, you could pick out all the records that match a certain site name string. You could select out records with specified method codes. You could get all records with inclinations that are positive.

Use **magic_select.py** to pick out all the best-fit lines based on AF demagnetization data from the *pmag_specimens.txt* file unpacked from the MagIC database in the download_magic.py example.

```
% magic_select.py -f pmag_specimens.txt -key magic_method_codes LP-DIR-AF has -F
76  records written to  ./AF_specimens.txt
  % magic_select.py -f AF_specimens.txt -key magic_method_codes DE-BFL has -F AF_
69  records written to  ./AF_BFL_specimens.txt
```

5.2.64 make_magic_plots.py

[MagIC] [make_magic_plots docs]

This program will inspect the files in the current directory and autogenerate some standard plots.

5.2.65 Measurement Import Scripts

2G_bin_magic.py

This program imports the binary format generated by the 2G proprietary software. It will place the orientation information in the er_samples.txt file and er_sites.txt file by default. Naming and orientation conventions can be specified as in the sections on Naming conventtions and Orientation conventions. You can also specify sampling method codes that pertain to ALL samples (see section on field information.)

AGM_magic.py

[Essentials Chapter 5] & [MagIC] [AGM_magic docs]

This program imports Micromag hysteresis files into magic_measurements formatted files. Because this program imports data into the MagIC database, specimens need also to have sample/site/location information which can be provided on the command line. If this information is not available, for example if this is a synthetic specimen, specify -syn SYN_NAME for synthetic on the command line instead of the -spn SPEC_NAME switch.

Someone named Lima Tango has measured a synthetic specimen named *myspec* for hysteresis and saved the data in a file named *agm_magic_example.agm*. The backfield IRM curve for the same specimen was saved in *agm_magic_example.irm*. Use the program **AGM_magic.py** to import the data into a magic_measurements formatted output file. These were measured using cgs units, so be sure to set the units switch properly. Combine the two output files together using combine_magic.py. [This can be plotted using **hysteresis_magic.py** or **quick_hyst.py**.] **hysteresis_magic.py** will calculate various hysteresis parameters and put them in the relevant magic tables for you.] s

```
% AGM_magic.py -spn myspec --usr "Lima Tango" -f agm_magic_example.agm -u cgs

results put in  ./agm_measurements.txt
% AGM_magic.py -spn myspec --usr "Lima Tango" -f agm_magic_example.irm -u cgs  -bak

results put in  ./irm_measurements.txt
% combine_magic.py -F magic_measurements.txt -f agm_measurements.txt irm_measurements.txt
```

CIT_magic.py

Craig Jones' **PaleoMag** software package (`http://cires.colorado.edu/people/jones.craig/PMag3.html`) imports various file formats, including the so-called 'CIT' format developed for the Cal Tech lab.The documentation for the CIT sample format is here: `http://cires.colorado.edu/people/jones.craig/PMag_Formats.html#SAM_format`. Demagnetization data for each specimen is in its own file in a directory with all the data for a study. These files are rather strictly formatted with fields determined by the character number in the line. A typical datafile (hat tip Swanson-Hysell et al. (2009)) has this format:

```
MP 18-1
 _________169.0  47.0 170.0  42.0   1.0
NRM    256.8 -60.3  63.7 -77.5 4.36E-03 001.8 351.8 -72.5 3.244495 13.18438 0.620
TT     254.0 -60.5  70.1 -77.5 4.28E-03 002.3 356.4 -72.5 4.752208 16.68586 1.969
AF     260.3 -60.5  56.5 -76.8 4.32E-03 000.6 346.6 -71.8 3.370241 2.506036 0.391
AF  20 256.4 -61.0  65.2 -76.8 4.33E-03 000.5 352.9 -71.8 3.260177 2.374987 0.277
AF  40 254.1 -61.3  70.0 -76.7 4.34E-03 000.8 356.4 -71.7 3.283688 5.443790 0.216
TT 100 254.6 -61.6  69.1 -76.4 4.36E-03 002.6 355.8 -71.3 1.512697 20.08180 0.568
TT 150 259.5 -61.8  59.9 -75.7 4.49E-03 000.6 349.0 -70.6 2.292811 4.155307 0.582
TT 200 254.2 -62.5  70.2 -75.4 4.57E-03 000.7 356.7 -70.4 3.548058 4.420838 1.117
TT 250 255.4 -62.5  68.0 -75.4 4.59E-03 000.9 355.1 -70.4 5.709379 4.016543 2.128
TT 300 254.3 -63.7  70.4 -74.3 4.55E-03 001.3 356.9 -69.2 4.430973 9.541593 1.820
TT 350 254.9 -64.5  69.6 -73.5 4.49E-03 000.8 356.3 -68.4 2.008315 6.203830 0.690
TT 400 252.8 -64.2  72.8 -73.8 4.48E-03 001.6 358.8 -68.8 2.923507 12.20715 1.171
```

There is an optional first line with 'CIT' on it (not shown here).

There must be a file with the suffix '.sam' in the same directory which gives details about the samples and a list of the datafiles in the directory. The .sam file has the form:

```
Mamainse Point
 47.1 275.3  -7.2
MP18-1
MP18-2
MP18-3
MP18-4
MP18-5
MP18-6
```

The first line is a comment, the second is the latitude and longitude followed by a declination correction.

For detailed description of the file format, check the PaleoMag home page.

Use the program **CIT_magic.py** to import the data files from the example data files in the *CIT_magic* directory of the *Measurement_Import* directory of the Datafiles_2.0 directory. The location name was "Mamainse Point", the naming convention was #2 and they were collected by drilling and with a magnetic compass.

```
 CIT_magic.py  -f bMP.sam -loc "Mamainse Point" -spc 0 -ncn 2 -mcd 'FS-FD:SO-MAG'
Mamainse Point first run (samples collected 10/05, samples run 04/06)

specimens stored in  ./er_specimens.txt
samples stored in  ./er_samples.txt
averaging: records  1 3
MP18-1 started with  23  ending with  21
Averaged replicate measurements
averaging: records  1 2
MP18-2 started with  25  ending with  24
Averaged replicate measurements
averaging: records  1 3
MP18-3 started with  26  ending with  24
Averaged replicate measurements
averaging: records  1 2
MP18-4 started with  25  ending with  24
Averaged replicate measurements
averaging: records  1 3
MP18-5 started with  26  ending with  24
Averaged replicate measurements
averaging: records  1 2
MP18-6 started with  25  ending with  24
Averaged replicate measurements
data stored in  ./magic_measurements.txt
```

These data can now be viewed and interpreted using, for example zeq_magic.py.

HUJI_magic.py

This program was developed for the late Prof. Hagai Ron at the Hebrew Universisty, Jerusalem for files with formats like this:

```
mk118 07R-08329 5/10/2007 7:35 N 0 10.4 73.9 10.4 73.9 2.23E-03
mk118 07R-08330 5/10/2007 7:37 A 5 359.2 60 359.2 60 2.19E-03
mk118 07R-08331 5/10/2007 7:38 A 10 358.4 48 358.4 48 1.50E-03
mk118 07R-08332 5/10/2007 7:40 A 15 357.7 43.5 357.7 43.5 8.34E-04
mk118 07R-08333 5/10/2007 7:42 A 20 358.4 42.5 358.4 42.5 4.89E-04
mk118 07R-08334 5/10/2007 7:44 A 25 358.8 40.6 358.8 40.6 3.06E-04
mk118 07R-08335 5/10/2007 7:46 A 30 356.6 41.1 356.6 41.1 2.16E-04
mk118 07R-08336 5/10/2007 7:48 A 40 357 39.3 357 39.3 1.39E-04
mk118 07R-08337 5/10/2007 7:50 A 50 357 40 357 40 1.03E-04
mk118 07R-08338 5/10/2007 7:53 A 60 351.7 46.9 351.7 46.9 9.64E-05
mk118 07R-08339 5/10/2007 7:55 A 70 350.7 48.7 350.7 48.7 8.89E-05
mk118 07R-08340 5/10/2007 7:58 A 80 353.8 38.8 353.8 38.8 6.77E-05
```

HUJI_magic.py works in a similar fashion to sio_magic.py.

LDEO_magic.py

[MagIC] [LDEO_magic docs]

This program works in much the same fashion as sio_magic.py but the file formats are different. Here is a typical example:

```
isaf2.fix
LAT:   .00  LON:    .00
    ID     TREAT  I  CD    J    CDECL CINCL  GDECL GINCL  BDECL BINCL  SUSC  M/V
-------------------------------------------------------------------------------
is031c2       .0  SD  0 461.600 163.9  17.5  337.1  74.5  319.1  74.4    .0   .0
```

The first line is the file name and the second has the latitude and longitude information. The columns are:

```
ID: specimen name
TREAT:  treatment step
I:  Instrument
CD:  Circular standard devation
J: intensity.  assumed to be total moment in 10^-4 (emu)
CDECL:  Declination in specimen coordinate system
CINCL:  Declination in specimen coordinate system
GDECL:  Declination in geographic coordinate system
GINCL:  Declination in geographic coordinate system
BDECL:  Declination in bedding adjusted coordinate system
BINCL:  Declination in bedding adjusted coordinate system
```

```
SUSC:  magnetic susceptibility (in micro SI)a
M/V: mass or volume for nomalizing (0 won't normalize)
```

Use the program to import the file *ldeo_magic_example.dat* into the MagIC format. It was an AF demagnetization experiment.

```
% LDEO_magic.py -f ldeo_magic_example.dat -LP AF
results put in  magic_measurements.txt
```

IODP_csv_magic.py

Just for fun, download the whole core data from IODP expedition 318, Site U1359, Hole A. (Section Half set to 'A') from the IODP LIMS database using WebTabular. These data were used in Tauxe et al. (2012) after some editing (removal core ends and disturbed intervals). Convert them into the MagIC format using **ODP_csv_magic.py**:

```
% IODP_csv_magic.py -f SRM_318_U1359_A_A.csv
processing:  SRM_318_U1359_A_A.csv
specimens stored in  ./er_specimens.txt
samples stored in  ./er_samples.txt
......
Averaged replicate measurements
data stored in  ./magic_measurements.txt
```

This takes a while so be patient....

If you also downloaded the core summary data from the LIMS database, you can plot the data and the core tops using core_depthplot.py

PMD_magic.py

This format is the one used to import .PMD formatted magnetometer files (used for example in the PaleoMac software of Cogné, 2003) into the MagIC format. (See `http://www.ipgp.fr/~cogne/pub/paleomac/PMhome.html` for the PaleoMac home page. The version of these files that **PMD_magic.py** expects (UCSC version) contains demagnetization data for a single specimen and have a format like this:

```
Strat post corrected - Summit Springs, -
ss0101a   a= 93.9   b= 61.0   s=  0.0   d=  0.0   v=11.0E-6m3  06/17/2003 12:00
STEP  Xc [Am2]  Yc [Am2]  Zc [Am2]  MAG[A/m]   Dg    Ig    Ds    Is  a95
```

```
H000 -7.46E-06 -1.61E-05 +5.31E-05 +5.09E+00  73.3  35.2  73.3  35.2  0.5  cryoSI
H002 -7.57E-06 -1.53E-05 +5.03E-05 +4.83E+00  73.2  35.7  73.2  35.7  0.7  cryoSI
H005 -7.35E-06 -1.16E-05 +3.62E-05 +3.52E+00  71.5  38.3  71.5  38.3  0.7  cryoSI
H008 -7.35E-06 -6.27E-06 +1.92E-05 +1.95E+00  68.5  47.1  68.5  47.1  0.7  cryoSI
H012 -6.84E-06 -3.32E-06 +1.20E-05 +1.29E+00  69.1  56.1  69.1  56.1  0.6  cryoSI
H018 -6.00E-06 -2.26E-06 +8.96E-06 +1.00E+00  69.3  60.5  69.3  60.5  0.6  cryoSI
H025 -5.16E-06 -1.74E-06 +7.18E-06 +8.19E-01  69.2  62.5  69.2  62.5  0.5  cryoSI
H050 -3.12E-06 -9.23E-07 +4.03E-06 +4.71E-01  69.3  64.6  69.3  64.6  0.5  cryoSI
H090 -1.80E-06 -5.32E-07 +2.23E-06 +2.65E-01  67.6  65.6  67.6  65.6  0.5  cryoSI
H180 -9.49E-07 -2.94E-07 +1.21E-06 +1.42E-01  67.7  64.8  67.7  64.8  0.5  cryoSI
```

The first line is a comment line. The second line has the specimen name, the core azimuth (a=) and plunge (b=) which are assumed to be the lab arrow azimuth and plunge (Orientation scheme #4)D. The third line is a header explaining the columns in the file.

Use **PMD_magic.py** to convert the file *ss0101a.pmd* in the directory 'PMD' in the 'PMD_magic' folder of the *Measurement_import* directory in the example datafiles directory. These were taken at a location named 'Summit Springs' and have a naming convention of the type XXXX[YYY], where YYY is sample designation with Z characters from site XXX, or

```
% PMD_magic.py -f ss0101a.pmd -loc "Summit Springs" -ncn 4-2 -spc 1 -mcd SO-MAG -
 results put in  ./magic_measurements.txt
sample orientations put in  ./er_samples.txt
```

Because each file must be imported separately, you should use a different name for the output file for each input file (otherwise you will overwrite the default each time) and set the switch for sample file to append for subsequent imports:

```
% PMD_magic.py -f ss0102a.pmd -loc "Summit Springs" -ncn 4-2 -spc 1 -mcd SO-MAG -
sample information will be appended to  ./er_samples.txt
results put in  ./magic_measurements.txt
sample orientations put in  ./er_samples.txt
```

After you finish importing all the data, combine the individual files together with combine_magic.py and look at them with, for example, zeq_magic.py.

(see also PMM_redo.py for importing interpretations made on these files using the UCSC (Jarboe) software).

sio_magic.py

[sio_magic docs]

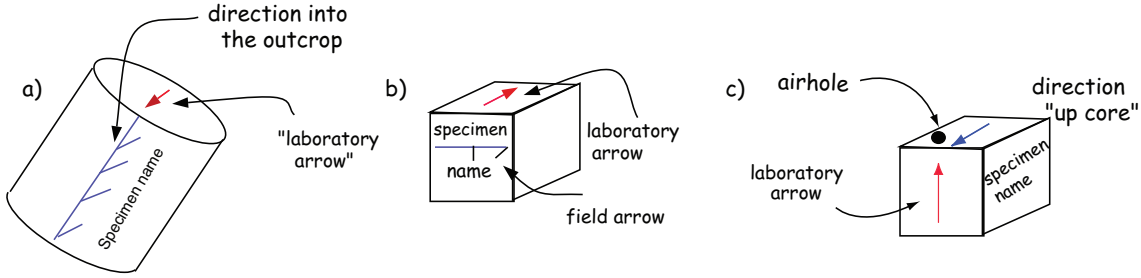
The program **sio_magic.py** allows conversion of the SIO format magnetometer files to the MagIC common measurements format. It allows various experiment types so read the help message. The SIO format is a space delimited file:

```
sc12b1      0.00        0.3      2.878E-2      185.0      9.3
sc12b1      5.00        0.3      2.864E-2      187.7      9.6
sc12b1      10.00       0.3      2.833E-2      185.5      10.0
sc12b1      15.00       0.2      2.771E-2      188.2      10.0
sc12b1      20.00       0.4      2.644E-2      187.6      10.1
sc12b1      30.00       0.2      2.182E-2      186.0      9.9
sc12b1      40.00       0.4      1.799E-2      185.3      9.9
sc12b1      50.00       0.2      1.570E-2      187.9      9.6
sc12b1      60.00       0.2      1.423E-2      185.6      10.0
sc12b1      80.00       0.1      1.275E-2      187.4      9.5
```

The columns are:

Specimen treatment intensity declination inclination optional_string

The treatment field is the temperature (in centigrade), the AF field (in mT), the impulse field strength, etc. For special experiments like IRM acquisition, the coil number of the popular ASC impulse magnetizer can be specified if the treatment steps are in volts. The position for anisotropy experiments or whether the treatment is "in-field" or in zero field also require special formatting. The units of the intensity field are in cgs and the directions are relative to the 'lab arrow' on the specimen. Here are some examples of commonly used specimens and conversions from field arrow to lab arrow.



As an example, we use data from Sbarbori et al. (2009) done on a set of samples from the location "Socorro", including AF, thermal, and thellier experimental data. These were saved in *sio_af_example.dat, sio_thermal_example.dat,*

and sio_thellier_example.dat respectively. The lab field for the thellier experiment was 25 μT and was applied along the specimen's Z axis (phi=0,theta=90).] Convert the example files into magic_measurement formatted files with names like *af_measurements.txt*, etc. Then combine them together with combine_magic.py:

```
Note:  all of these should actually be on ONE line:

% sio_magic.py -f sio_af_example.dat -F af_measurements.txt
    -LP AF -spc 1 -loc Socorro

averaging: records  11 13
averaging: records  14 16
averaging: records  17 19
averaging: records  20 22
sc12b1 started with  22  ending with  14
Averaged replicate measurements
results put in  af_measurements.txt


    % sio_magic.py -f sio_thermal_example.dat -F thermal_measurements.txt  \
    -LP T -spc 1 -loc Socorro
averaging: records  6 7
sc12b2 started with  23  ending with  22
Averaged replicate measurements
results put in  thermal_measurements.txt


% sio_magic.py -f sio_thellier_example.dat -F  thellier_measurements.txt  \
      -LP T -spc 1 -loc Socorro -dc 25 0 90
averaging: records  10 11
sc12b2 started with  56  ending with  55
Averaged replicate measurements
results put in  thellier_measurements.txt


% combine_magic.py -F magic_measurements.txt -f  thellier_measurements.txt   \
       thermal_measurements.txt  af_measurements.txt

File  ./thermal_measurements.txt  read in with  22  records
```

```
File  ./thellier_measurements.txt  read in with  55  records
File  ./af_measurements.txt  read in with  14  records
All records stored in  ./magic_measurements.txt
```

The data in these files can be plotted and interpreted with dmag_magic.py, zeq_magic.py, or thellier_magic.py depending on the experiment.

Note that there are more examples of data file formats and import schemes in the sections on anisotropy of anhysteretic and thermal remanences.

SUFAR4-asc_magic.py

[Essentials Chapter 13]; [MagIC] [SUFAR4-asc_magic docs]

The Agico Kappabridge instrument comes with the SUFAR program which makes the measurements and saves the data in a txt file like that in *SUFAR4-asc_magic_example.txt* in the *SUFAR4-asc_magic* directory. These data were measured on a KLY4S instrument with a spinning mode. Import them into the MagIC format:

```
%SUFAR4-asc_magic.py -f SUFAR4-asc_magic_example.txt -loc U1356A -spc 0 -ncn 5
anisotropy tensors put in  ./rmag_anisotropy.txt
bulk measurements put in  ./magic_measurements.txt
specimen info put in  ./er_specimens.txt
sample info put in  ./er_samples.txt
site info put in  ./er_sites.txt

        You can now import your data into the Magic Console and complete data entry,
        for example the site locations, lithologies, etc. plotting can be done
         with aniso_magic.py
```

TDT_magic.py

[Essentials Chapter 10] and [MagIC] [TDT_magic docs]

This program imports the default data format for the ThellierTool Program of Leonhardt et al. (2004). After importing, the data can be viewed with **thellier_magic.py**.

The file format of the ThellierTool .tdt format is:

```
Thellier-tdt
```

```
50.12 0 0 0 0
2396D 0.00 1821.50 4.6 49.4
2396D 200.00 1467.5 7.4 44.5
2396D 200.11 1808.3 4.3 34.4
2396D 200.13 1421.89 6.1 46
2396D 230.00 1317.69 6.9 45.3
2396D 230.11 1711.2 4.3 32.9
2396D 260.00 1150.89 8.4 44.7
2396D 200.12 1475.92 4.3 34.7
...
```

The first two lines are headers. The first column of the second line is the applied field in μT. The rest of this line is azimuth and plunge of the fiducial line and dip direction and dip of the bedding plane. The data columns are: specimen, treatment, intensity, declination and inclination. the intensity is the magnetization in 10^{-3} A/m. The treatment is of the form XXX.YY where XXX is the temperature and YY is 00, 11, 12, 13, 14 OR 0, 1, 2, 3, 4 where 0 is the NRM step, 11/1 is the pTRM acquisition step, 12/2 is the pTRM check step and 13/3 is the pTRM tail check. 14/4 is the additivity check.

Use the program **TDT_magic.py** to import the ThellierTool tdt formatted file *TDT_magic_example.dat* into the MagIC format. The field was 52.12 μT applied along the 0,0 direction.

```
% TDT_magic.py -f tdt_magic_example.dat -loc TEST -dc 50.12 0 0
results put in  magic_measurements.txt
```

5.2.66 measurements_normalize.py

[MagIC] [measurements_normalize docs]

This program takes specimen weights or volumes from an *er_specimen.txt* formatted file and normalizes the magnetic moment data to make magnetizations. Weights must be in kilograms and volumes in m^3.

Use the program **measurements_normalize.py** to generate weight normalized magnetizations for the data in the file *magic_measurements.txt* in the —it measurements_normalize directory. Use the specimen weights in the file *specimen_weights.txt*.

```
% magic_select.py -f magic_measurements.txt \
     -key magic_method_codes LP-IRM has -F irm_measurements.txt
205  records written to  ./irm_measurements.txt
```

```
% measurements_normalize.py -f irm_measurements.txt \
      -fsp specimen_weights.txt -F irm_norm_measurements.txt
Data saved in  ./irm_norm_measurements.txt
```

5.2.67 mk_redo.py

MagIC [mk_redo docs]

The programs zeq_magic.py and thellier_magic.py make *pmag_specimen* formatted files which can be used for further data reduction either by plotting or contributing to site means, etc. Sometimes it is useful to redo the calculation, using anisotropy corrections or a change in coordinate systems, etc. The re-doing of these specimen level calculations is handled by, for example zeq_magic_redo.py or thellier_magic_redo.py. These programs use *magic_measurements* formatted files and perform calculations as dictated by a "redo" file which has the specimen name, bounds for calculation and, in the case of the demagnetization data interpretation, the type of calculation desired (best-fit lines with directional estimation magic method code:DE-BFL, best-fit planes with those with magic method code DE-BFP, etc.).

Make "redo" files from the existing *pmag_specimen* formatted file in the data files downloaded from the MagIC website as in **download_magic.py** and examine them as follows:

```
% mk_redo.py
% cat zeq_redo
sr01a1 DE-BFL 473 823 A
sr01a2 DE-BFL 473 848 A
sr01c2 DE-BFL 473 823 A
sr01d1 DE-BFL 373 798 A
.......
% cat thellier_redo
sr01a1 573 823
sr01a2 573 848
sr01c2 673 823
sr01d1 373 823

.....
```

Note that the temperature steps are in kelvin and the AF demagnetization steps are in Tesla as required in the MagIC data base.

5.2.68 nrm_specimens_magic.py

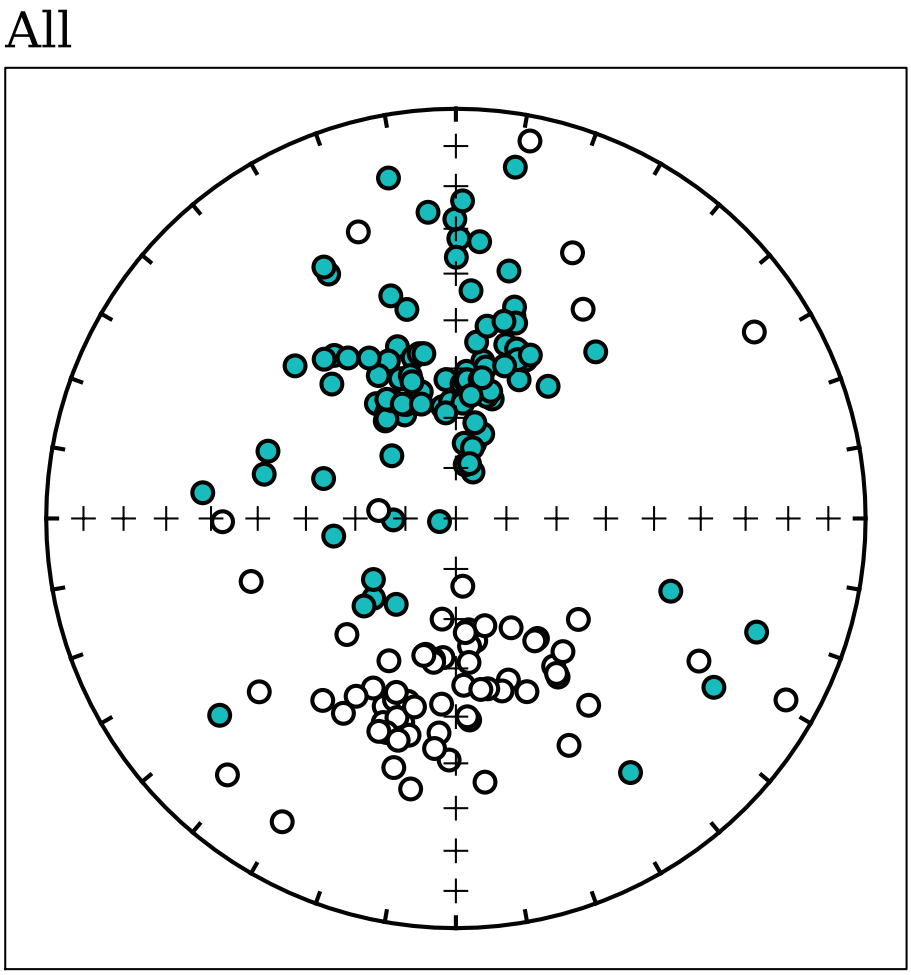
MagIC [nrm_specimens_magic docs]

After making NRM measurements, it is frequently useful to look at the directions in equal area projection to get a "quick look" at the results before proceeding to step wise demagnetization. The data in the magic_measurements files are usually in specimen coordinates - not geographic, so we need a way to rotate the data into geographic and or stratigraphic coordinates and save them in a pmag_specimens formatted file for plotting with **eqarea_magic.py**. The program **nrm_specimens_magic.py** will do this for you.

Get into the directory you made for tshe **download_magic.py** example. Use **nrm_specimens_magic.py** to convert the NRM measurements in *magic_measurements.txt* to geographic coordinates saved in a file named *nrm_specimens.txt*. The orientation data are in the file *er_samples.txt*. Then plot the specimen directions for the entire study using **eqarea_magic.py**:

```
% nrm_specimens_magic.py -crd g
er_samples.txt  read in with  271  records
Data saved in  nrm_specimens.txt

% eqarea_magic.py -f nrm_specimens.txt
177  records read from  ./nrm_specimens.txt
All
sr01a1 SO-SUN   324.1    66.0
sr01a2 SO-SUN   325.3    62.1
sr01c2 SO-SUN   344.7    63.8
sr01d1 SO-CMD-NORTH   327.8    64.5
sr01e2 SO-SUN   333.7    67.0
...
```

The first command created a file *nrm_specimens.txt* and the second created an equal area projection of the NRM directions in geographic coordinates which should look like this:



5.2.69 orientation_magic.py

[Essentials Chapter 9] and [Preparing for MagIC] [orientation_magic docs]

Try to import the file *orientation_example.txt* into the *er_samples.txt* and *er_sites.txt* files using **orientation_magic.py**. See Section 4.1.2 for details about the *orient.txt* file format. It has field information for a few sites. The samples were oriented with a Pomeroy orientation device (the default) and it is desirable to calculate the magnetic declination from the IGRF at the time of sampling (also the default). Sample names follow the rule that the sample is designated by a letter at the end of the site name (convention #1 - which is the default). So we do this by:

```
orientation_magic.py -f orient_example.txt
```

```
saving data...
Data saved in  ./er_samples.txt  and  ./er_sites.txt
```

5.2.70 parse_measurements.py

This program reads in the *magic_measurements.txt* file and creates an *er_specimens.txt* file. The specimen volumes and/or weights can then be put in columns labelled **specimen_volume** and **specimen_weight** respectively. Volumes must be in m^3 and weights in kg. (Yes you can do the math...).

Try this out on the *magic_measurements.txt* file created in the irmaq_magic.py example. Pretend you have a bunch of specimen weights you want to use to normalize the NRM with the program measurements_normalize.py.

```
% parse_measurements.py
site record in er_sites table not found for:  318-U1359A-002H-1-W-109
site record in er_sites table not found for:  318-U1359A-002H-2-W-135
site record in er_sites table not found for:  318-U1359A-002H-3-W-45
site record in er_sites table not found for:  318-U1359A-002H-4-W-65
.....
```

The program appears a bit flustered because you have no *er_sites.txt* file in this directory. If you DID, you would overwrite whatever site name was in that file onto the specimen table. This allows you to carry the changes in that table through to the specimen table (see orientation_magic.py.)

5.2.71 pca.py

[Essentials Chapter 11] [pca docs]

This program calculates best-fit lines, planes or Fisher averages through selected treatment steps. The file format is a simple space delimited file with specimen name, treatment step, intensity, declination and inclination. Calculate the best-fit line through the first ten treatment steps in data file *zeq_example.txt*:

```
 % pca.py -dir L 1 10 -f zeq_example.dat
eba24a DE-BFL
0 0.00 339.9 57.9 9.2830e-05
1 2.50 325.7 49.1 7.5820e-05
...
eba24a DE-BFL 10    2.50  70.00    8.8   334.9    51.5
```

According to the help message, this is: specimen name, calculation type, N, beg, end, MAD, declination and inclination. The calculation type is the MagIC method code for best-fit lines (see Essentials Appendix **??**.)

5.2.72 plotXY.py

[plotXY docs]

This program makes a simple XY plot from any arbitrary input file. You can specify which columns are X and Y, bounds on the columns, symbol color and size, axis labels and other options. See igrf.py for an example.

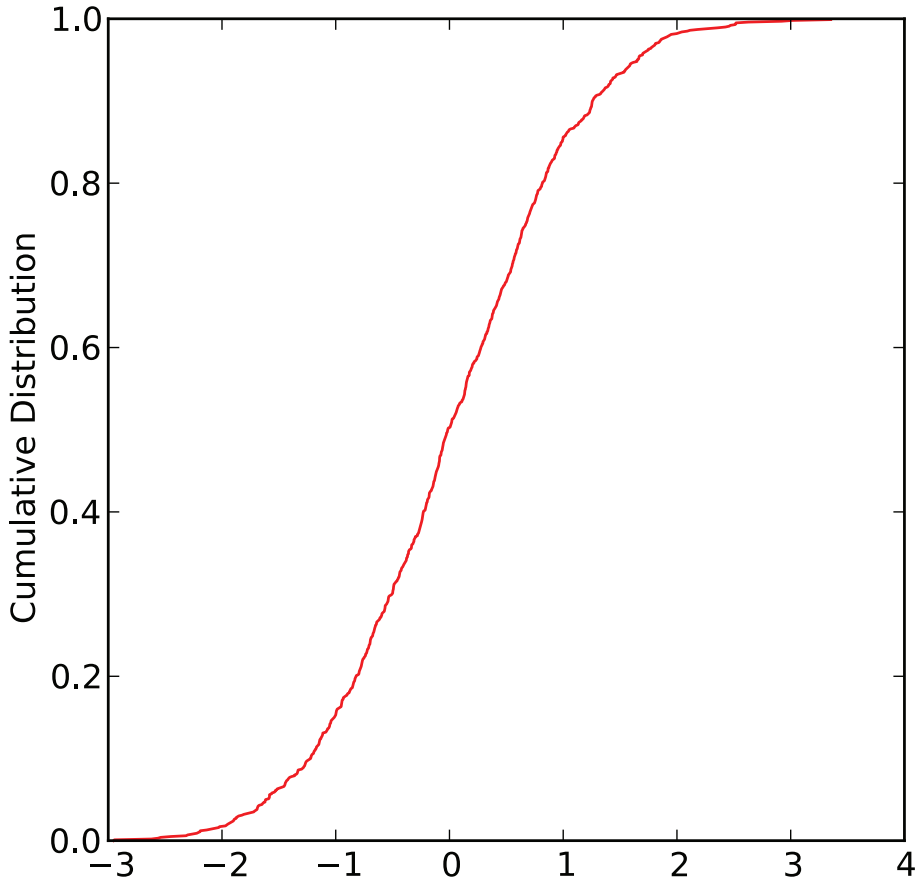
5.2.73 plot_cdf.py

[plot_cdf docs]

This program plots cumulative distribution functions of a single column of input data. Use as an example, a normally distributed set of 1000 data points generated by gaussian.py. Use the defaults of zero mean with a standard deviation of 1.

```
% gaussian.py -n 1000 > gaussian.out
  % plot_cdf.py -f gaussian.out
S[a]ve  plot, <Return> to quit a
1  saved in  CDF_.svg
```

which should have generated a plot something like this:



5.2.74 plot_magic_keys.py

MagIC [plot_magic_keys docs]

This program will plot any column in a specified MagIC formatted file against any other column in the same file.

5.2.75 plot_mapPTS.py

high resolution instructions [plot_mapPTS docs]

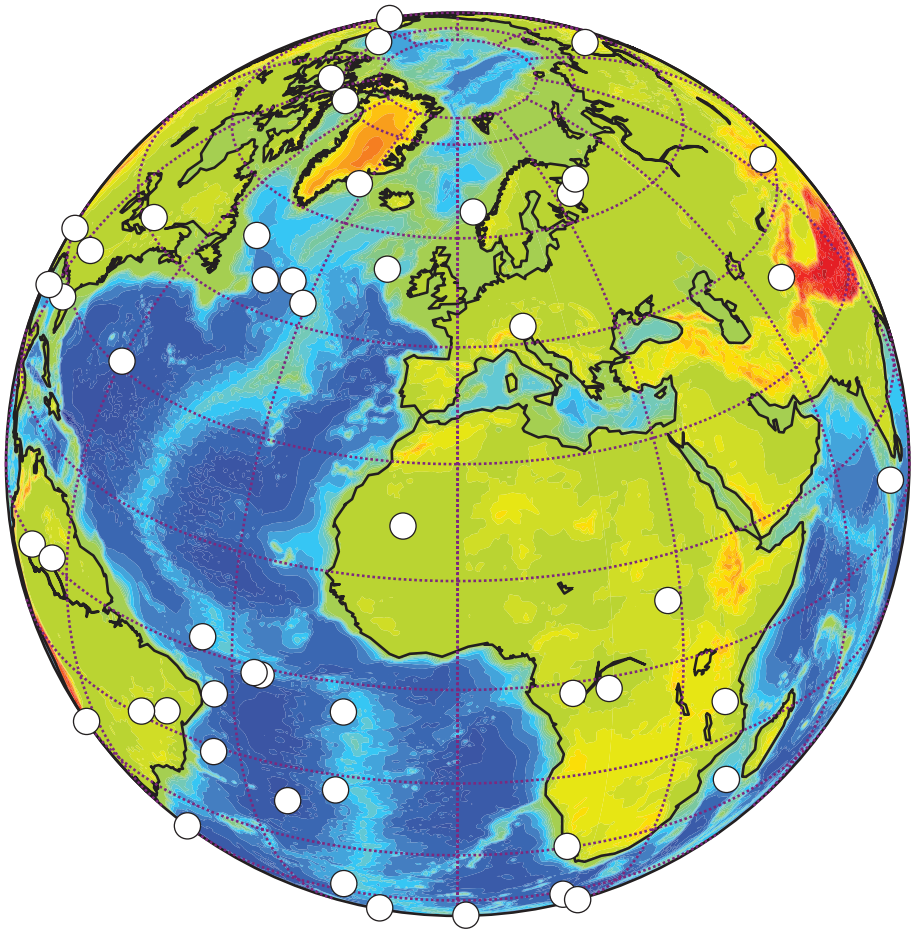
NOTE: This program only works if you have installed basemap (full version of Enthought Python as opposed to the free version.) It will not work with the free version of Enthought Python. **plot_mapPTS.py** will generate a simple map of the data points in a file (lon lat) on the desired projection. If you want to use high resolution or the etopo20 meshgrid (-etp option), you must install the high resolution continental outlines as described on the

high resolution instructions website. There are many options, so check the documentation (-h option) for details.

Draw a set of 200 uniformly distributed points on the globe with the program uniform.py. Plot these on an orthographic projection with the viewing point at a longitude of 0 and a latitude of 30°N. If you have installed the high resolution data sets, use the -etp option to plot the topographic mesh. Plot the points as large (size = 10) white dots. Also note that for some reason the .svg output does not place nicely with illustrator.

```
% uniform.py -n 200 >uniform.out
% plot_mapPTS.py -f uniform.out -prj ortho -eye 30 0 -etp -sym wo 10
please wait to draw points
 S[a]ve to save plot, Return to quit:  a
1  saved in  Map_PTS.pdf
```

which should produce a plot similar to this:



5.2.76 plotdi_a.py

[Essentials Chapter 11] [plotdi_a docs]

Place the following declination, inclination α_{95} data in a space delimited file called
plotdi_a_example.dat.

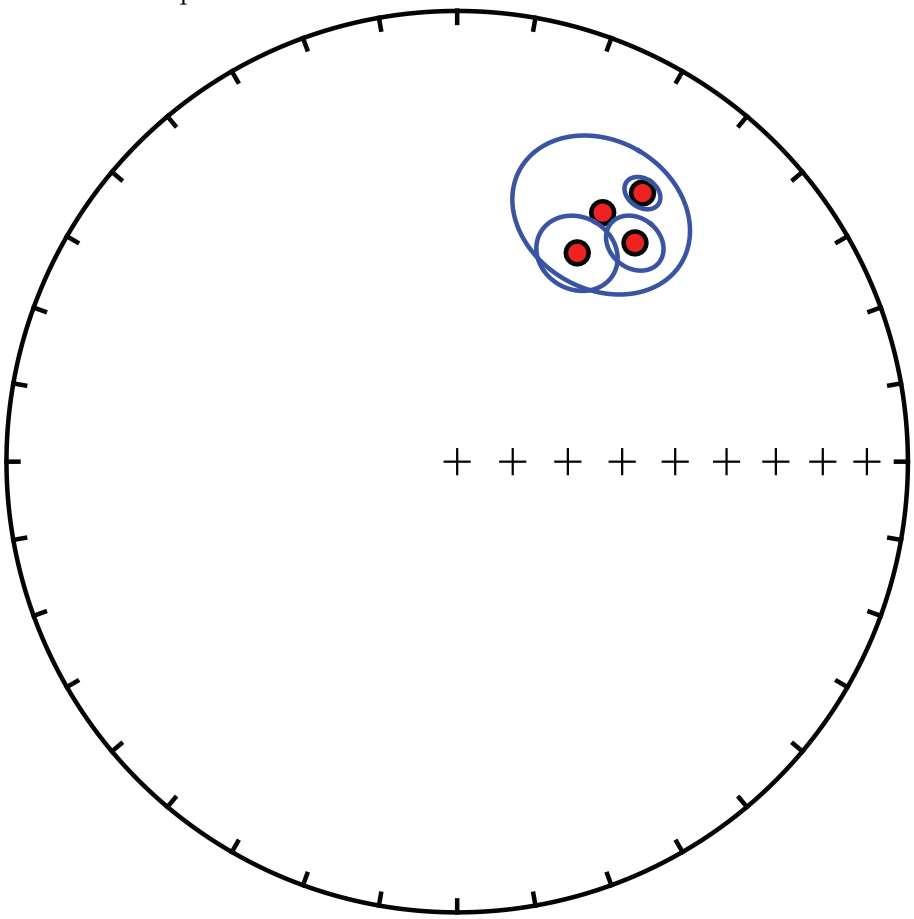
Dec	Inc	α_{95}
39.1	37.5	5.0
30.3	36.2	15
29.9	45.6	7
34.6	28.4	3

Make a plot of these data using **plotdi_a.py**:

```
% plotdi_a.py -f plotdi_a_example.dat
```

```
 S[a]ve to save plot, [q]uit, Return to continue:  a
1  saved in  eq.svg
```

which makes the plot:



5.2.77 pmag_results_extract.py

[:MagIC] [pmag_results_extract docs]

This program extracts a tab delimited txt file from a *pmag_results* formatted file. This allows you to publish data tables that have identical data to the data uploaded into the MagIC database. Try this out in the directory created for the **download_magic.py** example:

```
% pmag_results_extract.py -f pmag_results.txt
data saved in:  ./Directions.txt ./Intensities.txt ./SiteNfo.txt
```

This creates tab delimited files that can be incorporated into a paper, for example. You can also export the information in LaTeX format if you prefer.

5.2.78 pt_rot.py

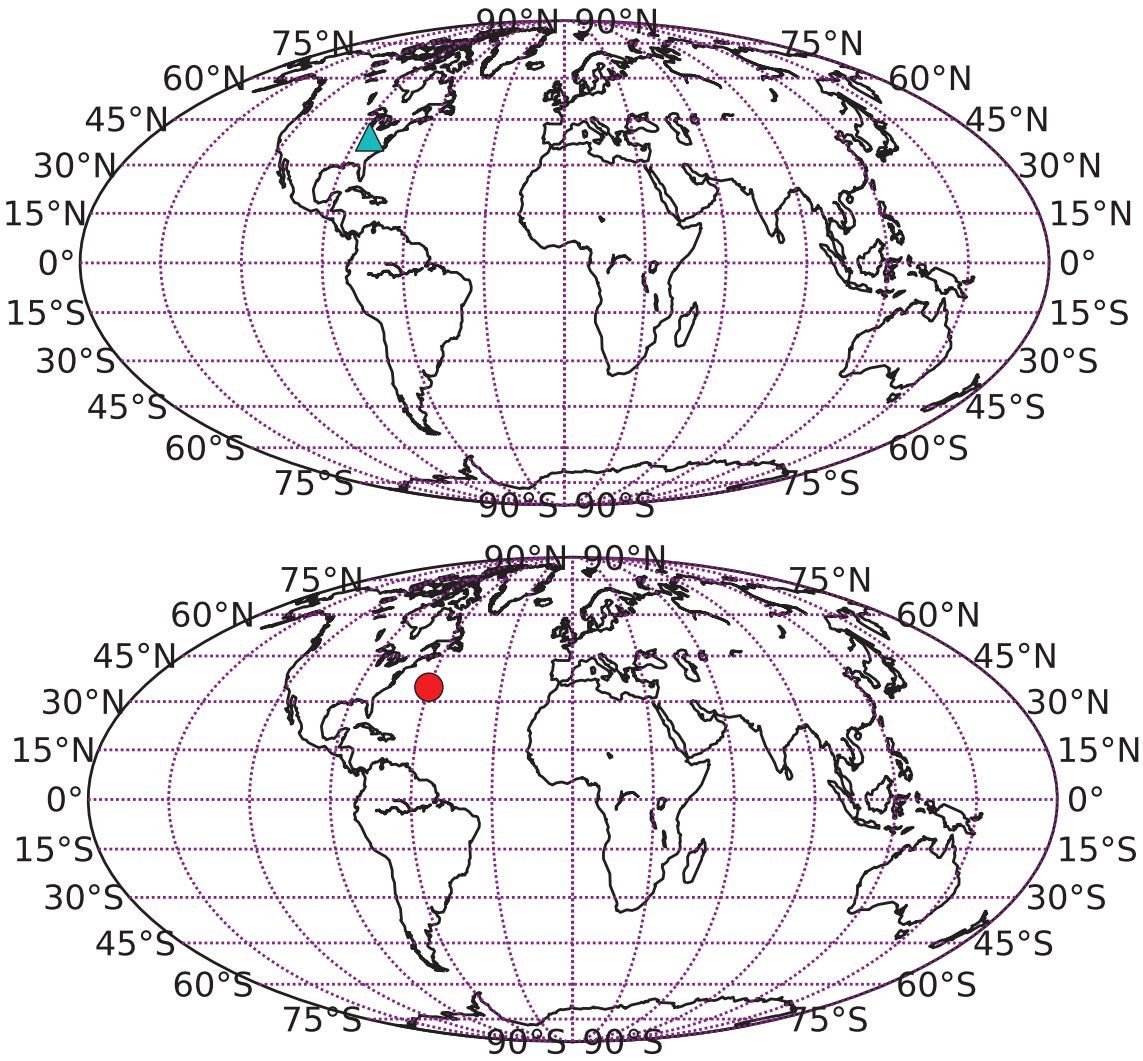
[Essentials Chapter 16] [pt_rot docs]

This program reads in a file with location, age and destination plate and rotates the data into the destination plate coordinates using the rotations and methods in Essentials Appendix A.3.5. Alternatively, you can supply your own rotation parameters with the -ff option.

First, save the location of Cincinnati (39.1 latitude, -84.7 longitude) in a file called *pt_rot.input* using either the UNIX **cat** function or your favorite text editor. Save the data as a lon/lat pair separated by a space. Then use the program **pt_rot.py** to rotate Cincinnati to (Northwest) African coordinates at 80 Ma. Plot both points using plot_mapPTS.py. You should save the lon, lat, plate, age, destination_plate information in a file called *pt_rot_example.dat* in the following:

```
% plot_mapPTS.py -f pt_rot.input -sym g^ 10
please wait to draw points
 S[a]ve to save plot, Return to quit:  a
1  saved in  Map_PTS.pdf
% pt_rot.py -f pt_rot_example.dat
299.763594614 34.6088989286
% pt_rot.py -f pt_rot_example.dat >pt_rot.out
% plot_mapPTS.py -f pt_rot.out -sym ro 10
please wait to draw points
 S[a]ve to save plot, Return to quit:  a
1  saved in  Map_PTS.pdf
```

The two plots will look like these:

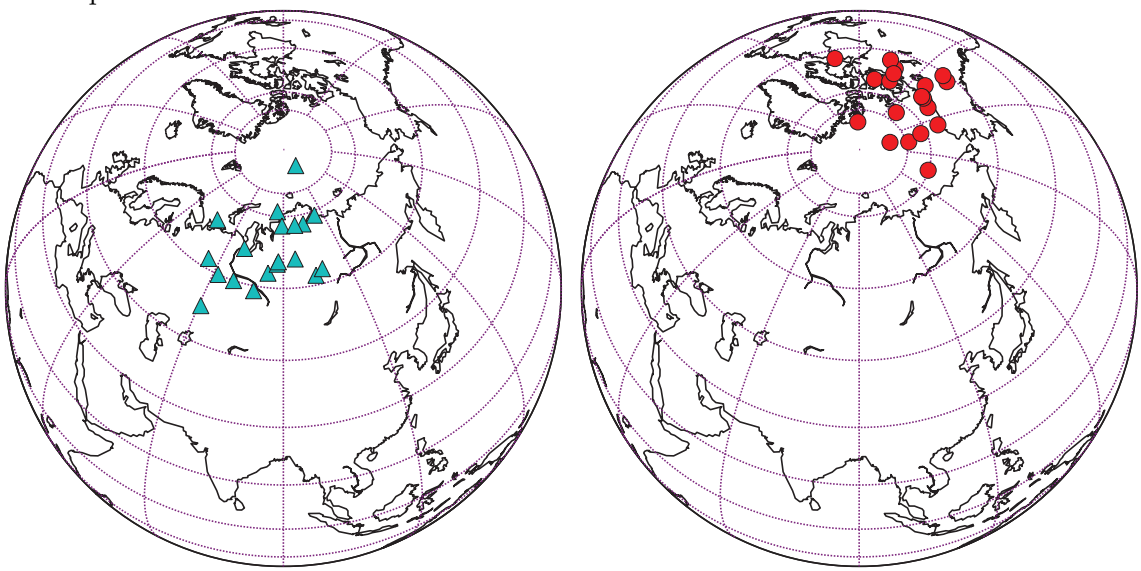


Now, use the program to rotate a selection of North American poles from 180-200 Ma (in the file *nam_180-200.txt* in the *pt_rot* directory) to Pangea A coordinates (finite rotation pole and angle in *nam_panA.frp*. Note that plot_mapPTS.py reads in longitude/latitude pairs, while **pt_rot.py** reads in latitude longitude pairs.

```
% plot_mapPTS.py -f nam_180-200.txt -prj ortho -eye 60 90 -sym c^ 10
please wait to draw points
 S[a]ve to save plot, Return to quit:  a
1  saved in  Map_PTS.pdf
```

```
% pt_rot.py -ff nam_180-200.txt nam_panA.frp
165.005966833 75.1055653383
272.368103052 83.1542552681
281.233199772 63.3738811186
260.36973769 70.7986591555
250.91392318 70.3067026787
251.438099128 65.9369002914
255.371917029 63.4115373574
204.92422073 71.7944043036
.....
% pt_rot.py -ff nam_180-200.txt nam_panA.frp > pt_rot_panA.out
% plot_mapPTS.py -f pt_rot_panA.out -prj ortho -eye 60 90 -sym ro 10
please wait to draw points
 S[a]ve to save plot, Return to quit:  a
1  saved in  Map_PTS.pdf
```

These plots should look like this:



5.2.79 qqlot.py

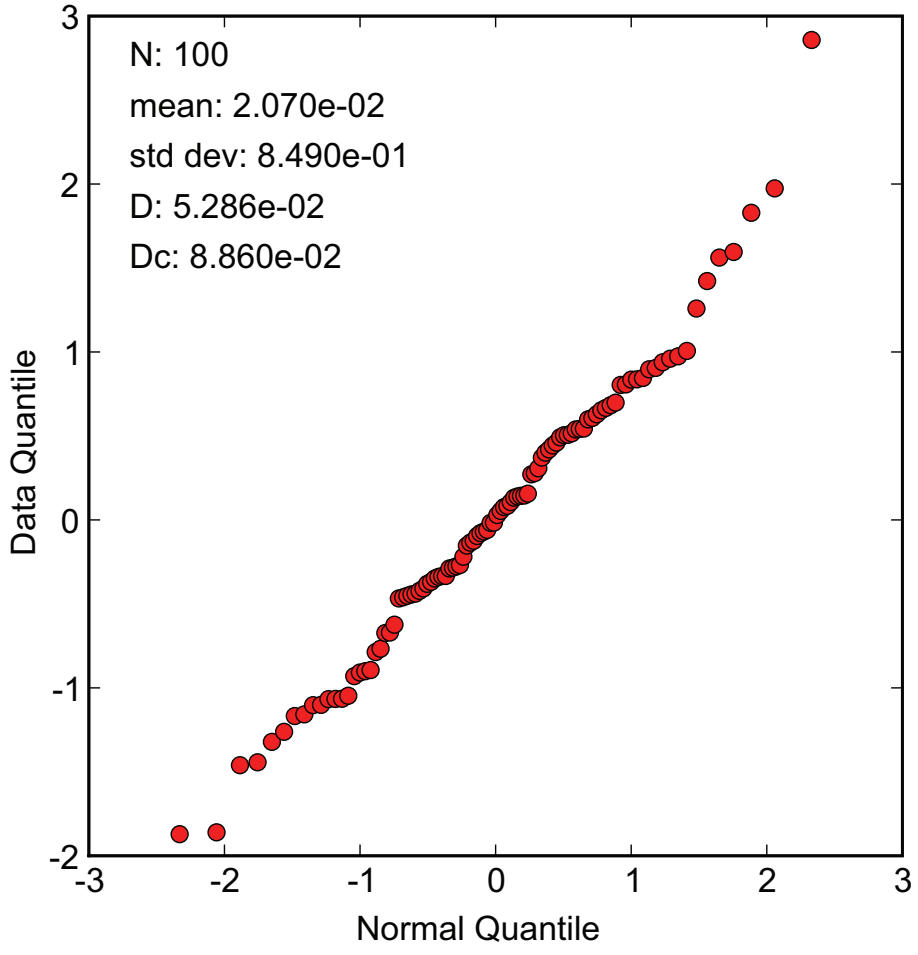
[Essentials Appendix B.1.5] [qqplot docs]

Makes a quantile-quantile plot of the input data file against a normal distribution. The plot has the mean, standard deviation and the D statistic as well as the D_c statistic expected from a normal distribution. Use

qqplot.py to test whether the data generated with **gaussian.py** is in fact normally distributed. (It will be 95% of the time!).

```
% gaussian.py  -F  gauss.out
% qqplot.py -f gauss.out
mean,sigma, d, Dc
0.02069909 0.849042146783 0.0528626896977 0.0886
 S[a]ve to save plot, [q]uit without saving:  a
1  saved in  qq.svg
```

which generates this plot:



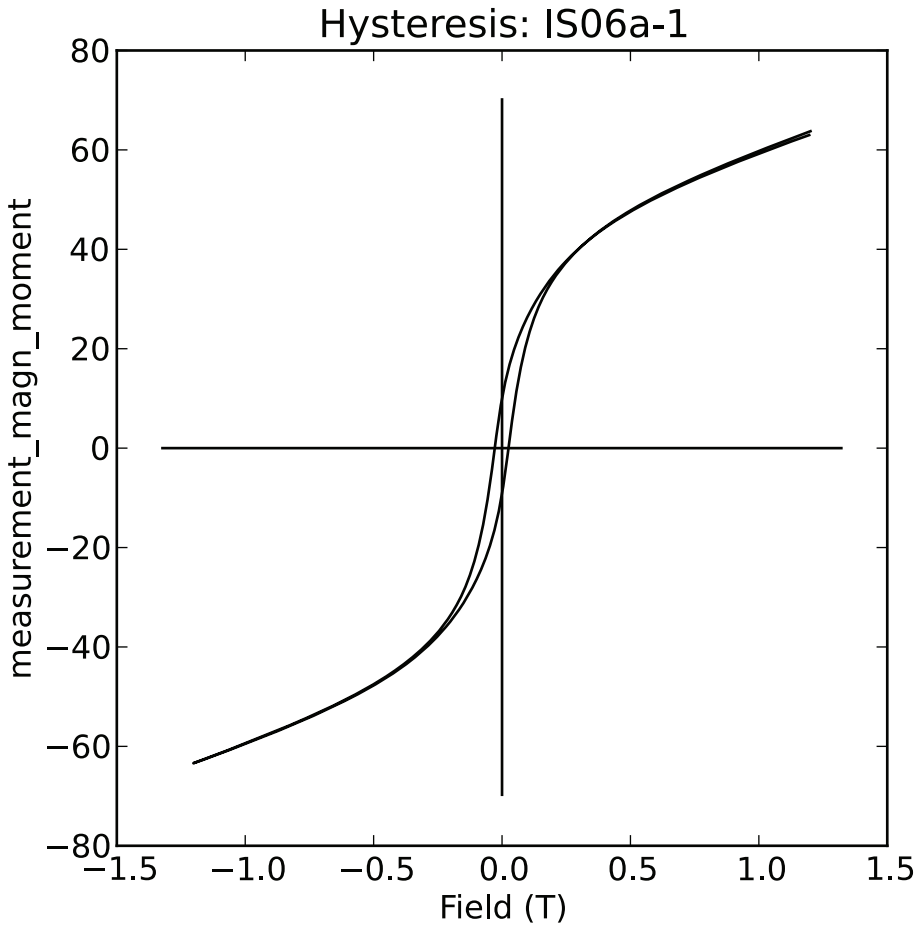
5.2.80 quick_hyst.py

[Essentials Chapter 5] and [MagIC] [quick_hyst docs]

hysteresis_magic.py makes plots of hysteresis loops and calculates the main hysteresis parameters. For a quick look with no interpretation, you can use **quick_hyst.py**. Try it out on the data file *hysteresis_magic_example.dat* in the *hysteresis_magic* directory.

```
% quick_hyst.py -f hysteresis_magic_example.dat -fmt svg
IS06a-1 1 out of  8
working on t:  273
S[a]ve plots, [s]pecimen name, [q]uit, <return> to continue
 a
1  saved in  LO:__SI:_IS06_SA:_IS06a_SP:_IS06a-1_TY:_hyst_.svg
IS06a-1 1 out of  8
working on t:  273
S[a]ve plots, [s]pecimen name, [q]uit, <return> to continue
 q
Good bye
```

which makes a plot like this:



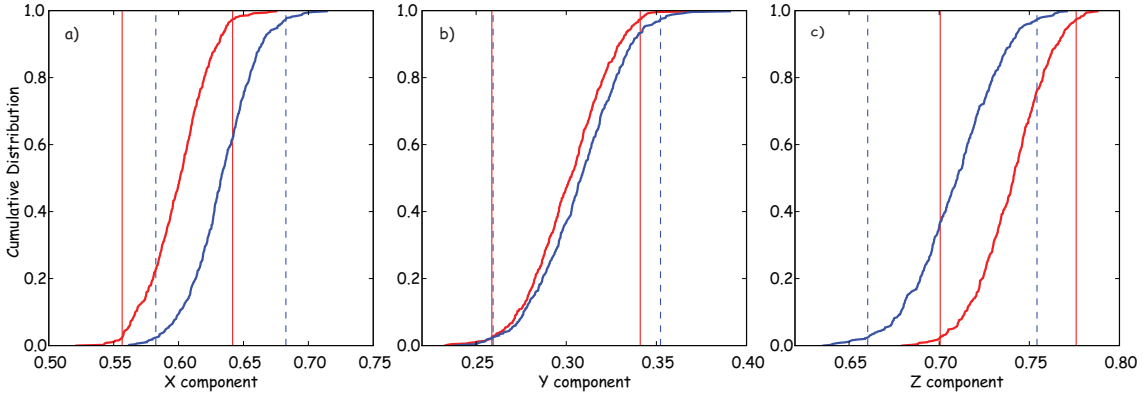
5.2.81 revtest.py

[Essentials Chapter 12] [revtest docs]

Use **revtest.py** to test whether the two modes in the data set *revtest_example.txt* are antipodal or not:

```
% revtest.py -f revtest_example.txt
doing first mode, be patient
doing second mode, be patient
s[a]ve plots, [q]uit: a
2  saved in  REV_Y.svg
1  saved in  REV_X.svg
3  saved in  REV_Z.svg
```

which produces these plots:



Because the 95% confidence bounds for each component overlap each other, the two directions are not significantly different.

5.2.82 revtest_magic.py

[Essentials Chapter 12] and MagIC [revtest_magic docs]

Same as revtest.py but for *pmag_sites* MagIC formatted files. Try it out on the data file *revtest_sites.txt*. Then try using customize_criteria.py to change or create a *pmag_criteria.txt* file that fits your needs and redo the reversals test using only the selected sites.

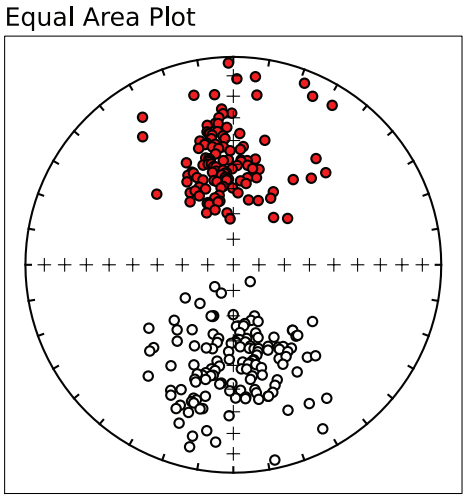
```
 revtest_magic.py -f revtest_magic_example.txt
doing first mode, be patient
doing second mode, be patient
s[a]ve plots, [q]uit: q
% revtest_magic.py -f revtest_sites.txt -exc
```

5.2.83 revtext_MM1990.py

This program tests whether two sets of observations (one with normal and one with reverse polarity) could have been drawn from distributions that are 180 ° apart using the McFadden and McElhinny (1990) implementation of the Watson (1983) V statistic test for a common mean. The implementation of the V statistic test in this program is the same as in WatsonsV.py. Use **revtext_MM1990.py** to perform a reversal test on data from the Ao et al., 2013 study of Early Pleistocene fluvio-lacustrine sediments of the Nihewan Basin of North China.

Lets plot the combined data set *Ao_etal2013_combined.txt* using eqarea.py:

```
% eqarea.py -f Ao_etal2013_combined.txt
```

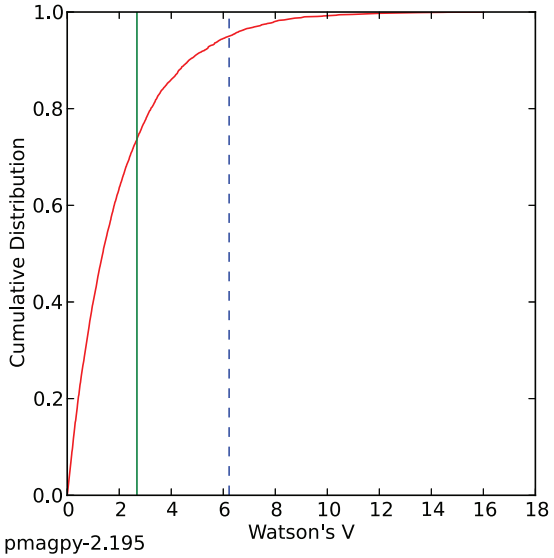


Conduct a reversal test between the normal directions *Ao_etal2013_norm.txt* and reversed directions *Ao_etal2013_rev.txt*:

```
% MM1990_revtest.py -f Ao_etal2013_norm.txt -f2 Ao_etal2013_rev.txt
be patient, your computer is doing 5000 simulations...

Results of Watson V test:

Watson's V:           2.7
Critical value of V:  6.0
"Pass": Since V is less than Vcrit, the null hypothesis that the
 two populations are drawn from distributions that share a
 common mean direction (antipodal to one another) cannot be rejected.

M&M1990 classification:

Angle between data set means: 2.9
Critical angle for M&M1990:   4.3
The McFadden and McElhinny (1990) classification for this test is: 'A'
```

As with WatsonsV.py, this program produces a plot that displays the cumulative distribution (CDF) of the Watson V values resulting from the

Monte Carlo simulation. In the plot below, the red line is the CDF, the green solid line is the value of the V statistic calculated for the two populations and the dashed blue line is the critical value for the V statistic. In this example, the data are consistent with the two directional groups sharing a common mean as evidenced by V (green solid line) being a lower value than the critical value of V (dashed blue line).



5.2.84 s_eigs.py

[Essentials Chapter 13] [s_eigs docs]

Convert the .s format data in *s_eigs_example.dat* to eigenvalues and eigenvectors:

```
% s_eigs.py -f s_eigs_example.dat
0.33127186 239.53  44.70 0.33351338 126.62  21.47 0.33521473  19.03  37.54
0.33177859 281.12   6.18 0.33218277 169.79  73.43 0.33603862  12.82  15.32
0.33046979 283.57  27.30 0.33328307 118.37  61.91 0.33624712  16.75   6.13
...
```

5.2.85 s_geo.py

[Essentials Chapter 13] [s_geo docs]

Rotate the .s data into geographic coordinates using **s_geo.py**. The input format is the 6 tensor elements of s, followed by the azimuth and plunge of the X axis.

```
% s_geo.py -f s_geo_example.dat
0.33412680 0.33282733 0.33304587 -0.00015289 0.00124843 0.00135721
0.33556300 0.33198264 0.33245432 0.00087259 0.00024141 0.00096166
0.33584908 0.33140627 0.33274469 0.00131845 0.00118816 0.00002986
...
```

5.2.86 s_hext.py

[Essentials Chapter 13] [s_hext docs]

Take the output from the **s_geo.py** example and calculate Hext statistics:

```
% s_geo.py -f s_geo_example.dat | s_hext.py
F =  5.79 F12 =  3.55 F23 =  3.66
N =  8  sigma =  0.000641809950
0.33505     5.3    14.7    25.5   124.5    61.7    13.3   268.8    23.6
0.33334   124.5    61.7    25.1   268.8    23.6    25.5     5.3    14.7
0.33161   268.8    23.6    13.3     5.3    14.7    25.1   124.5    61.7
```

5.2.87 s_tilt.py

[Essentials Chapter 13] [s_tilt docs]

Rotate the .s data saved in *s_tilt_example.dat* into stratigraphic coordinates:

```
% s_tilt.py -f s_tilt_example.dat
0.33455709 0.33192658 0.33351630 -0.00043562 0.00092778 0.00105006
0.33585501 0.33191565 0.33222935 0.00055959 -0.00005316 0.00064731
0.33586669 0.33084923 0.33328408 0.00142267 0.00013233 0.00009202
0.33488664 0.33138493 0.33372843 -0.00056597 -0.00039086 0.00004873
.
.
```

5.2.88 s_magic.py

[MagIC] [MagIC] [s_magic docs]

Import .s format file output from the **s_tilt.py** example into an *rmag_anisotropy* formatted file. Files of the *rmag_anisotropy* format can be plotted with **aniso_magic.py**. To see how this works, use the program **s_magic.py** as follows:

```
% s_tilt.py -f s_tilt_example.dat >s_magic_example.dat
% s_magic.py -f s_magic_example.dat
data saved in  ./rmag_anisotropy.txt
```

This creates the output file *rmag_anisotropy.txt* by default, which can be plotted with the program aniso_magic.py.

5.2.89 scalc.py

[Essentials Chapter 14] [scalc docs]

Calculate the S scatter statistic for a set of VGPs saved in *scalc_example.txt*. Repeat using a Vandamme (Vandamme et al., 1994) variable cutoff. Then get the bootstrap bounds on the calculation.

```
% scalc.py -f scalc_example.txt
99    19.5     90.0
% scalc.py -f scalc_example.txt -v
89    15.2     32.3
% scalc.py -f scalc_example.txt -v -b
89    15.2    13.3     16.8    32.3
```

Using no cutoff, the VGP scatter was 19.5°. The Vandamme co-latitude cutoff was 32.3° which threw out 6 points and gave a scatter of 15.2°.

5.2.90 scalc_magic.py

[Essentials Chapter 14] [scalc_magic docs]

This is the same as **scalc.py** but works on *pmag_results* formatted files. Try it out on the *pmag_results.txt* file in the directory created for the **download_magic.py** example. Use a VGP co-latitude cutoff of 30°.

```
% scalc_magic.py -f pmag_results.txt  -c 30
13    17.8     30.0
```

5.2.91 site_edit_magic.py

[site_edit_magic docs]

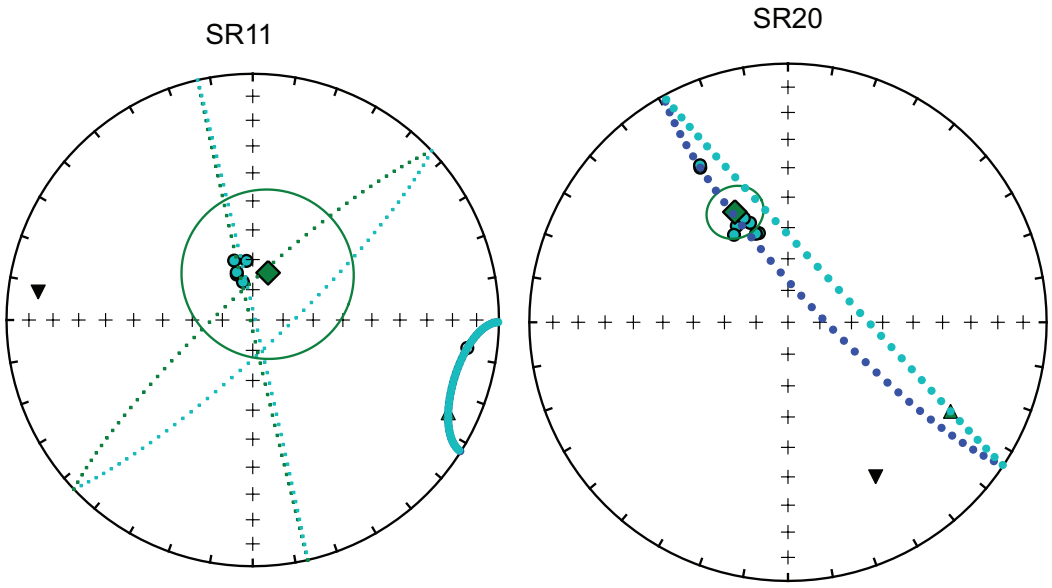
There are frequently outlier directions when looking at data at the site level. Some paleomagnetists throw out the entire site, while some arbitrarily discard individual samples, assuming that the orientations were 'bad'. Lawrence et al. (2009) suggested a different approach in which common causes of misorientation are specifically tested. For example, if the arrow indicating drill direction (see orientation conventions) was drawn the wrong way on the sample in the field, the direction would be off in a predictable way. Similarly (and more commonly) extraneous marks on the sample were used instead of the correct brass mark, the directions will fall along a small circle which passes through the correct direction. The program **site_edit_magic.py** plots data by site on an equal area projection, allows the user to select a particular specimen direction and plots the various pathological behaviors expected from common misorientations. If the stray directions can reasonably be assigned to a particular cause, then that sample orientation can be marked as 'bad' with a note as to the reason for exclusion and the directions from that sample can be excluded from the site mean, and the reason can be documented in the MagIC database.

Try this out using the data downloaded in the download_magic.py example.

```
% site_edit_magic.py
....
sr11
specimen, dec, inc, n_meas/MAD,| method codes
sr11a1:   340.6    74.1 8 / 0.9 | LP-DIR-AF:DE-BFL
sr11b1:   353.9    70.3 10 / 2.1 | LP-DIR-AF:DE-BFL
sr11c1:   342.6    69.4 8 / 1.3 | LP-DIR-AF:DE-BFL
sr11e2:   341.5    73.4 8 / 0.9 | LP-DIR-AF:DE-BFL
sr11f1:   136.6     9.4 14 / 3.5 | LP-DIR-T:DE-BFP
sr11g2:   345.5    77.0 8 / 0.8 | LP-DIR-AF:DE-BFL
sr11i1:    77.1     0.8 12 / 3.5 | LP-DIR-AF:DE-BFP
sr11j3:    97.4   -13.2 8 / 11.8 | LP-DIR-AF:DE-BFL

 Site lines planes  kappa   a95   dec   inc
sr11 6  2       5     28.4     18.0     73.6  6.7445
s[a]ve plot, [q]uit, [e]dit specimens, [p]revious site, <return> to continue:
 e
```

```
Enter name of specimen to check: sr11j3
Triangle: wrong arrow for drill direction.
Delta: wrong end of compass.
Small circle:  wrong mark on sample. [cyan upper hemisphere]
Mark this sample as bad? y/[n]  n
Mark another sample, this site? y/[n]  n
.....
sr20
no orientation data for  sr20j
specimen, dec, inc, n_meas/MAD,| method codes
sr20c1:   339.1    56.2 9 / 2.1 | LP-DIR-T:DE-BFL
sr20d2:   341.9    60.3 9 / 1.7 | LP-DIR-AF:DE-BFL
sr20e1:   332.1    55.3 10 / 1.0 | LP-DIR-AF:DE-BFL
sr20f1:   337.1    54.0 9 / 3.7 | LP-DIR-T:DE-BFL
sr20g1:   340.1    60.2 9 / 1.6 | LP-DIR-T:DE-BFL
sr20i1:   330.4    31.9 11 / 0.6 | LP-DIR-AF:DE-BFL
sr20i2:   330.7    31.1 9 / 2.3 | LP-DIR-T:DE-BFL

 Site lines planes  kappa   a95   dec   inc
sr20 7  0      38      9.9    335.1     50.0  6.8434
s[a]ve plot, [q]uit, [e]dit specimens, [p]revious site, <return> to continue:
 e
Enter name of specimen to check: sr20i1
Triangle: wrong arrow for drill direction.
Delta: wrong end of compass.
Small circle:  wrong mark on sample. [cyan upper hemisphere]
Mark this sample as bad? y/[n]  y
Reason: [1] broke, [2] wrong drill direction, [3] wrong compass direction,
[4] bad mark, [5] displaced block [6] other 2
Mark another sample, this site? y/[n]  n
```

Compare behaviors for sr11, sample sr11XXX and sr20, sample sr20i:



The latter plausibly fits the "wrong mark" hypothesis, while the former does not fit any of the common pathologies. The sample orientation in the *er_samples.txt* file has a note that it is 'bad' and will be ignored by the PmagPy programs when calculating site means.

5.2.92 specimens_results_magic.py

MagIC [specimens_results_magic docs]

Starting from the magnetometer output files, paleomagnetists interpret the data in terms of vectors, using software like zeq_magic.py or thellier_magic.py. The interpretations from these programs are stored in pmag_specimen formatted files. Once a pmag_specimens format file has been created, the data are usually averaged by sample and/or by site and converted into V[A]DMs and/or VGPs and put in a pmag_results formatted file along with the location and age information that are available. Data must be selected or rejected according to some criteria at each level (for example, the specimen MAD direction must be less than some value or the site κ must be greater than some value). This onerous task can be accomplished using the program **specimens_results_magic.py**. This program has many optional features, so the reader is encouraged just to look at the documentation.

Test it out by using the datafiles created in the site_edit_magic.py example. Block the interpretation of some of the samples owing to 'bad' orientation. Change the selection criteria using customize_criteria.py. Then generate a new *pmag_specimen.txt* file using mk_redo.py, zeq_magic_redo.py and combine_magic.py. Then do the averaging by site with **specimens_results_magic.py**. The age bounds of the data are 0-5 Ma (-age 0 5 Ma). Use the existing (or modified) criteria (-exc option). There are some paleointensity data, so use the current site latitude for the VADM calculation (-lat option). Calculate the data in the geographic coordinate system (-crd g). Then extract the data into tab delimited tables using pmag_results_extract.py

```
% mk_redo.py
% zeq_magic_redo.py -crd g
Processing  173  specimens - please wait
Recalculated specimen data stored in  ./zeq_specimens.txt
% combine_magic.py -F pmag_specimens.txt -f zeq_specimens.txt thellier_specimens
File  ./zeq_specimens.txt  read in with  173  records
File  ./thellier_specimens.txt  read in with  50  records
All records stored in  ./pmag_specimens.txt
% specimens_results_magic.py -age 0 5 Ma -exc -lat -crd g
Acceptance criteria read in from  pmag_criteria.txt

 Pmag Criteria stored in  pmag_criteria.txt

 sites written to  pmag_sites.txt
 results written to  pmag_results.txt
% pmag_results_extract.py -fa er_ages.txt
data saved in:  ./Directions.txt ./Intensities.txt ./SiteNfo.txt
```

Note that this procedure is more or less automated in the MagIC.py GUI.

5.2.93 stats.py

[Essentials Chapter 11] [stats docs]

Calculates gaussian statistics for sets of data. Calculate the mean of the data generated in the gaussian.py example and saved in *gauss.out*:

```
% stats.py -f gauss.out
100 10.12243251 1012.243251 2.79670530387 27.6287868663
```

which according to the help message is:

```
N, mean, sum, sigma, (%) , stderr, 95% conf.
where sigma is the standard deviation
where % is sigma as percentage of the mean
stderr is the standard error and
95% conf.=  1.96*sigma/sqrt(N)
```

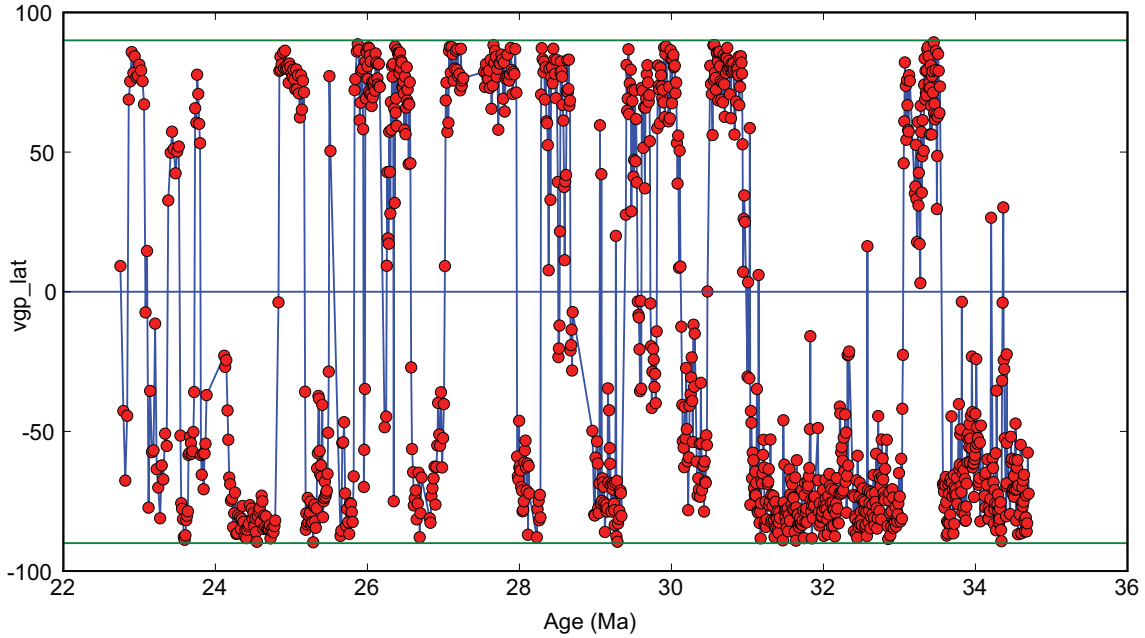
5.2.94 strip_magic.py

[Essentials Chapter 15] and [MagIC]

Follow the instructions for download_magic.py but search for Tauxe and Hartl and 1997. Download the smartbook (text file) and unpack it into a new directory using the **download_magic.py** command and the file *zmab0094214tmp02.txt* as the input file (in the Tauxe and Hartl directory). First run **strip_magic.py** to see what is available for plotting, then plot the inclination data versus depth (pos). Then plot the VGP latitudes versus age:

```
% download_magic.py -f zmab0094214tmp03.txt
working on:  'er_locations'
er_locations  data put in  ./er_locations.txt
working on:  'er_sites'
er_sites  data put in  ./er_sites.txt
working on:  'er_samples'
..........
% strip_magic.py
available objects for plotting:  ['all']
available X plots:  ['age', 'pos']
available Y plots:  ['lon', 'int', 'lat']
available method codes:  ['LP-PI-IRM', 'LP-PI-REL']
% strip_magic.py -x pos -y int -mcd LP-PI-IRM
 S[a]ve to save plot, [q]uit without saving:  a
1  saved in  strat.svg
 %strip_magic.py -x age -y lat
 S[a]ve to save plot, [q]uit without saving:  a
1  saved in  strat.svg
```

The last command made this plot:



5.2.95 sundec.py

[Essentials Chapter 9] [sundec docs]

Use the program **sundec.py** to calculate azimuth of the direction of drill. You are located at 35° N and 33° E. The local time is three hours ahead of Universal Time. The shadow angle for the drilling direction was 68° measured at 16:09 on May 23, 1994.

The program **sundec.py** works either by interactive data entry or by reading from a file.

Save the following in a file called *sundec_example.dat*:

```
3 35 33 1994 5 23 16 9 68
```

which is:

Δ_{GMT} lat lon year mon day hh mm shadow_angle

We can analyze this file with either:

```
% sundec.py -f sundec_example.dat
  154.2
```

or

```
% sundec.py < sundec_example.dat
  154.2
```

or by manual input:

```
% sundec.py -i
Time difference between Greenwich Mean Time (hrs to ADD to
      GMT for local time):
<cntl-D> to quit 3
Year:  <cntl-D to quit> 1994
Month:  5
Day:  23
hour:  16
minute:  9
Latitude of sampling site (negative in southern hemisphere): 35
Longitude of sampling site (negative for western hemisphere): 33
Shadow angle: 68
  154.2
Time difference between Greenwich Mean Time (hrs to ADD to
      GMT for local time):
<cntl-D> to quit ^D
 Good-bye
```

In any case, the declination is 154.2°.

5.2.96 thellier_GUI.py

[Essentials Chapter 10] and [MagIC] [Thellier_GUI_tutorial.pdf]

The program **thellier_gui.py** combines functions from thellier_magic.py and new tools described by Shaar and Tauxe (2012) in a user-friendly graphical user interface (GUI). It can also be called from within MagIC.py, using files already prepared in the MagIC Project Directory and the interpretations from **thellier_gui.py** can be imported into the **MagIC.py** Project Directory after exiting the program. This section is a brief introduction on how to use **thellier_gui.py** as a stand alone application. For complete documentation, see:

`http://earthref.org/PmagPy/pmagpydocs/Thellier_GUI_tutorial.pdf`

A complete list of the definitions for paleointensity statistics used by Thellier_GUI.py is available here:

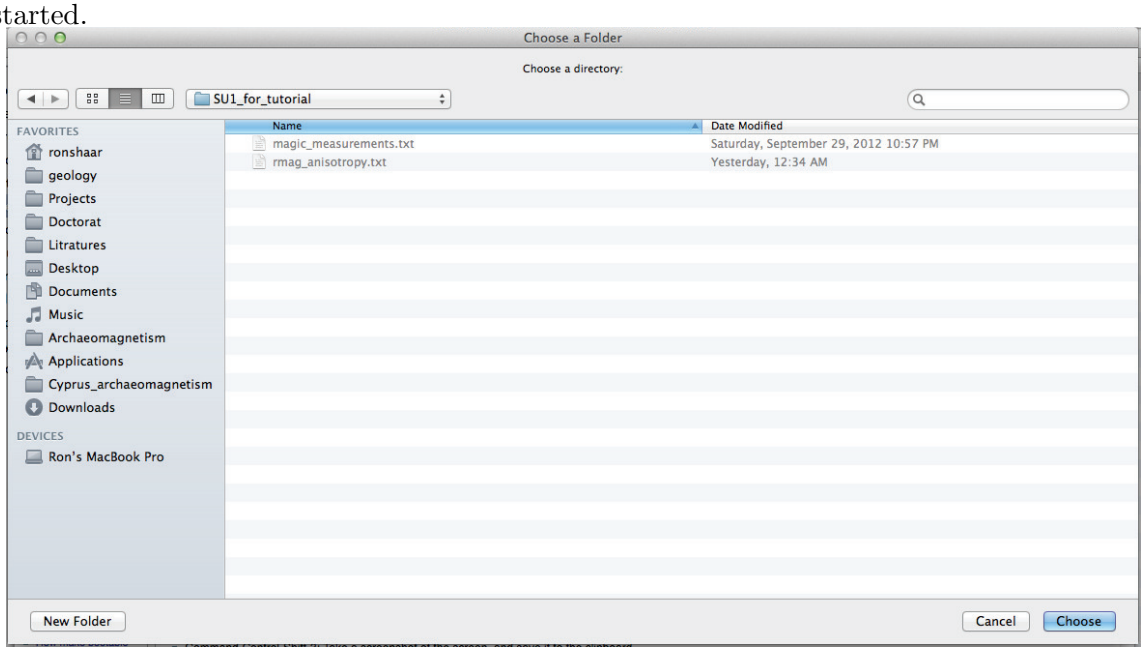
`http://earthref.org/PmagPy/pmagpydocs/Thellier_GUI_Paleointensity_Statistics_Definitions.pdf`

Starting the GUI

The GUI can be opened by typing in the Terminal window:

```
% thellier_gui.py
```

A "choose project directory" dialog window will appear as soon as the GUI is started.



The Project Directory should include a file named "magic_measurements.txt" . To make this use the GUI to Import your data files and combine your measurements. Also, if a file named "rmag_anistropy.txt" exists, then the program reads the anisotropy data from *rmag_anistropy.txt* . Reading and processing the measurements files may take several seconds, depending on the number of the specimens.

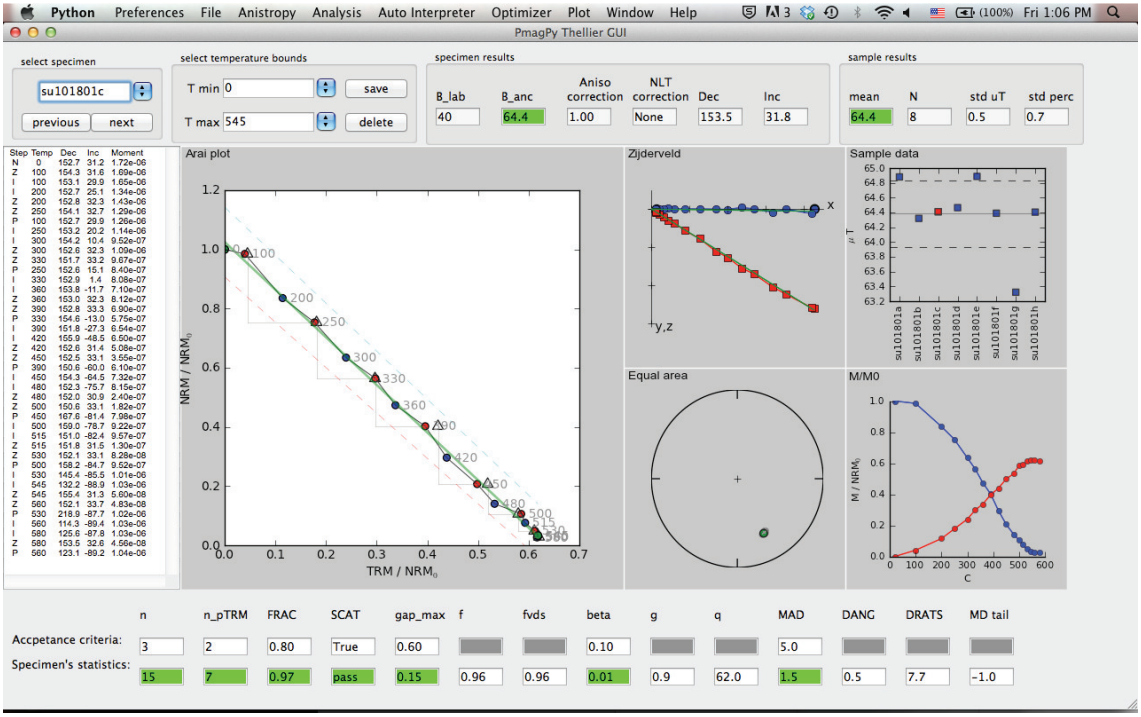
Reading and compiling measurements data

When the MagIC project directory is selected, the program reads all the measurement data, checks them, processes them and sorts them. If Non-linear-TRM (NLT) data exist in *magic_measurement.txt* then the program tries to process the data using Equations (1)-(3) in **?**. The program reads

magic_measurement.txt, and processes the measurements for presentation as Arai and Zijderveld plots. We recommend that you check all the warnings and errors in *Thellier_GUI.log* before starting to interpret the data. For details about warnings and error messages during these steps, consult the tutorial document: Thellier_GUI_tutorial.pdf. Also, consult the Preferences to change certain plotting options.

Main panel

This figure shows a snapshot of the main panel.



The top field in the panel includes the following buttons/controls (from left to right):

- **Specimen:** a list of the specimens in the project folder sorted by name.
- **previous/next:** buttons to move forward and backward in the specimens list.
- **T min/T max:** buttons to select temperature bounds.

- **save/delete:** save or delete current interpretation. This information will be used later to generate a redo file.
- **B_lab:** laboratory field in units of μT.
- **B_anc:** specimen's paleointensity in units of μT.
- **Aniso Correction:** anisotropy correction factor.
- **NLT Correction:** Non-Linear TRM (NLT) correction factor.
- **Dec/Inc:** Specimen declination/inclination calculated by PCA of the NRM in the selected temperature bounds.
- Sample mean, number of specimens, standard deviation and percent standard deviation.

The center of the main panel has these elements:

- **measurements text panel:** Four columns of the measurement data: Step is "N" for NRM, "Z" for zero field step, "I" for infield step, "P" for pTRM check, and "T" for tail check. The temperature of each step is given in C. Also shown declination, inclination and moment (in units of Am^2)
- **Arai plot:** Arai plot normalized by NRM_0. blue circles are zero field steps, red circles are infield steps, triangles are pTRM checks, blue squares are tail checks. Temperatures are displayed near data points. Temperature bounds and best fit line are marked in green. 'SCAT box' is marked with dashed lines (only if SCAT is True).
- **Zijderveld plot:** A Zijderveld plot of the NRM step. The x axis is rotated to the direction of the NRM, blue is the x-y projection, and red is x-z projection.
- **Equal area plot:** An equal area projections of the NRM steps. solid circles are positive inclination. open circles are negative inclinations.
- **Moment-temperature plot:** NRMs are in blue, pTRMs are in red.
- **sample data:** If at least two specimens have a saved interpretation, then their values are displayed on this plot. The mean $\pm$ standard deviation of the mean are marked as horizontal lines.

The bottom of the main panel include paleointensity statistics. The first line has the threshold values (empty if N/A). The second line is the specimen's statistics. For details see Appendix A1 in Shaar and Tauxe (2012).

Tutorial by example

If you have not already done so, download the example data files for PmagPy from
`http://earthref.org/PmagPy/Datafiles_2.0.zip`
There are two folder located under the folder *thellier_gu*: SU1_example and Tauxe_2006_example.

Case study 1: SU1_example:

Open a new Terminal window and fire up the program on the command line:

```
%thellier_gui.py
```

A choose directory dialog window will appear. Select the SU_1 project directory and click on the "choose" button.

- Manual interpretation Manual interpretation is the conventional approach of selecting temperature bounds for each specimen. Here, as example, we will interpret one sample, su100601, manually.
  - Set acceptance criteria: From the menu-bar choose Analysis → Acceptance criteria → Change acceptance criteria. A criteria dialog box will appear.
  - In the first line (specimen's statistics) set the following values: n=3, int_ptrm_n=2, SCAT=false (unchecked), gap_max=0.5, f_vds=0.75, beta=0.1, MAD=6, DRATS=20. delete the values in the other boxes (empty boxes).
  - In the second line (Sample's acceptance criteria) set the following values: int_n=3. delete the value in the other boxes.
  - In the third line (Sample mean calculation method: STDEV-OPT) check the Enable/Disable box, and set the following values: int_sigma=6, int_sigma_perc=10. Delete values in the other boxes. Figure 3 shows how it should look. Click on the OK button. A message dialog will appear telling you the criteria that you chose are saved in a file name pmag_criteria.txt. The next time you will open the GUI in this project directory, these criteria will automatically be chosen. Click OK on the message box.

- Manual interpretation of sample su100601: choose specimens su100601a from the specimen list. choose temperature bounds from the Tmin/Tmax windows: 200 and 580. B_anc shows 61.7 in green, which means that this interpretation passes the specimen's acceptance criteria. Now, change Tmax to 530. B_anc window change color to red, and also the fvds window (the row on the bottom of the GUI). This is an indication that the interpretation did not meet the criterion for fvds. Change Tmax to 580 (as before) and press "save" button. The interpretation in saved.
- Click on the next button to choose the next specimen (su100601b). Before analyzing this specimen, press on the "previous" button, so you can see that your previous interpretation was saved. click on "next" to return again to specimen su100601b. Find temperature bounds that pass the criteria (for example 200,580). Make sure that all colors are green, and click on the "save" button. Go over all the other specimens in the sample (su10060a to su100601g). you will find that three specimens can pass: a,c, and g. Look at the Sample data figure (top right) it shows the saved interpretation for all the specimens in the sample, and the mean ± standard deviation of the mean.
- To save all your interpretation into files, choose from the menubar Analysis → Save current interpretation. Three files will be generated in the project directory:
  - thellier_GUI.redo (the temperature bounds for each specimen):

    ```
    su100601a 473 853
    su100601c 473 853
    su100601g 473 853
    ```

    Note that the temperature bounds are in Kelvin.
  - *thellier_GUI.specimens.txt*: (the paleointensity statistics for each specimen - according to the headers. It is easy to view the content of this file using Excel):

    ```
    su100601a su100601 61.7 200 580 40.0 1.07 1.01 15 8
    0.77 0.87 0.89 0.14 0.02 N/A 0.63 N/A 5.11 4.14 PASS
    su100601c su100601 54.4 200 580 40.0 0.94 N/A 15 8 0.60
    0.78 0.79 0.23 0.01 N/A 5.19 N/A 2.51 1.33 PASS
    su100601g su100601 58.4 200 580 40.0 1.00 1.01 15 8
    0.70 0.83 0.85 0.19 0.02 N/A 2.80 N/A 3.12 1.96 PASS
    ```

* *thellier_GUI.samples.txt:* (the paleointensity statistics of the samples):

er_sample_name	sample_int_n	sample_int_uT	sample_int_sigma_uT	sample_int_sigma_perc
su100601	3	58.2	3.0	5.1

- You can save the Arai plot (or any other plots) from the File menu. Choose file → Save Plot → Save Arai plot. A file dialog window will appear. You can change the format of the figure in the bottom tab (pdf, svg, or png), but you can also save as other format by adding the appropriate suffix to the file name.
- close the GUI: File → Quit.
- Re-open the GUI as before (by writing the command "thellier_gui.py" in the terminal window) and choose the same working directory (SU1_example). Notice now that the criteria that you set before are automatically used. To import your previous interpretations and continue working on this dataset choose from the menu bar Analysis → import previous interpretation. Choose thellier_GUI.redo and click on the 'open' button.

- Automatic interpretation - *thellier_auto_interpreter:* Instead of trying to find manually the most appropriate temperature bounds, the **thellier_auto_interpreter.py** program allows a fast and consistent way to choose the temperature bounds for the specimens. For details see Shaar and Tauxe (2012).
  - We will use the same acceptance criteria.
  - Choose from the menu bar Auto interpreter → Run Thellier auto interpreter. The interpreter will run approximately 30 seconds. When its done, a message window will appear saying that the interpreter finished successfully (if not, check for errors in the terminal window, or in a file name *thellier_interpreter.log* located in a folder named *thellier_interpreter*. If the issue cannot be resolved send an e-mail with the description of the problem to rshaar@ucsd.edu). Click OK in the message window.
  - The automatic interpretations are automatically saved, and we can review them on the Arai plots. Let's look, for example, at the sample we interpreted manually: su100601. Choose specimen su100601a from the specimens list. At first glance, the choice seems strange because the automatic interpretation chose temperature bounds 0 and 580. A choice between 200 and 580 may seem

more reasonable as it provides an nearly straight line. The sample data figure explains this issue. The **thellier_auto_interpreter.py** program chooses the interpretations that minimize the standard deviation of the sample's mean. Six specimens from this sample passed the criteria, and the value of su100601a is much higher than the remaining five. So, the thellier_auto_interpreter chose the 'acceptable' interpretation with the shallowest slope. Go over the other specimens in the sample, and review the automatic interpretations. If you want, you can change the automatic interpretation, and save your own "manual" interpretation by choosing Analysis → Save current interpretation. Note that the output of **thellier_auto_interpreter.py** is sensitive to the choice of the acceptance criteria, and different criteria may lead to significantly different final results.

- Choosing the optimal criteria - *thellier_optimizer_2d:* Shaar and Tauxe (2012) discuss a method to choose the optimal specimen's acceptance criteria. Here, we follow the example given in this paper. Shaar and Tauxe (2012) suggested using seven statistics for specimen's acceptance criteria. The choice of four of which is trivial, and the remaining three are MAD, FRAC and β. The thellier_optimizer_2D helps finding the optimal values for FRAC and β. The thellier_optimizer_2D required three types of inputs: 'fixed_criteria", "test_groups", and "test_samples", as described below:

  - The test groups are defined in an "er_sample" file. An example for this file is given in working folder SU-1 named "optimizer_test_groups.txt". Open this file in Excel to understand its syntax. The first line is a general header. The rest of the file include sample name, group name, and comments.
  - To run the optimizer choose from the menu bar Optimizer → Run Thellier optimizer. click OK in the message dialog. The first step is setting the 'fixed_criteria". We choose for the specimen's criteria: n=3, int_ptrm_n=2,SCAT=checked, gap_max=0.6, MAD=5.0. We choose for the sample's criteria:int_n=3, int_n_outlier_check=6, EnableSTDEV-OPT, int_sigme_uT=5, int_sigma_perc=6. The rest of the boxes should be empty. click OK. click OK on the message window that will appear.
  - The thellier_optimizer_2D dialog window will appear. The first two rows of controls for choosing the range for β and FRAC.

choose beta from 0.05 to 0.15 in steps of 0.1 and FRAC from 0.7 to 0.9 in step of 0.02.
Inert the following functions to the text panel:
study_sample_n
max_group_int_sigma_uT
((max_group_int_sigma_uT <= 5) or (max_group_int_sigma_perc <= 6)) and int(study_sample_n)
click on the "Check function syntax" button. If there are no typos, then the text box below the button should be PASS.
click on the "Choose optimizer group file" button. A file dialog window will appear. choose the file "optimizer_test_groups.txt". Click the button "Run Optimizer" to start. The runtime using these parameter is about 8 minutes. ‘

– All the output files of the optimizer are saved in a folder named "optimizer" (see section 4.4). Compare the figures in the 'pdf' folder with Figure 7 in Shaar and Tauxe (2012).
– The figure in optimization_function_2.pdf suggest using $\beta \leq 0.10$ and FRAC ≥ 0.79. Set these values as acceptance criteria, and run thellier_auto_interpreter.

5.2.97 thellier_magic.py

[Essentials Chapter 10] and [MagIC] [thellier_magic docs]

To see how this works, download the text file from a recent paper by Shaar et al. (2011) from the MagIC database:

`http://earthref.org/MAGIC/m006899dt20120629115216/`

Unpack the data with download_magic.py. The original interpretations are stored in a file called *pmag_specimens.txt*, but **thellier_magic.py** looks for a specimen file called *thellier_specimens.txt*. To create such a file, use the program mk_redo.py followed by thellier_magic_redo.py. Then you can take a look at the data using thellier_magic.py and change some of the interpretations, for example adding one for the specimen s2s0-01 which was rejected by Shaar et al. (2011). This has both non-linear TRM acquisition and ATRM anisotropy data, so the corrections will have to be applied using **thellier_magic_redo.py** after exiting the thellier_magic.py program.

Here is a transcript of a session that does all this:

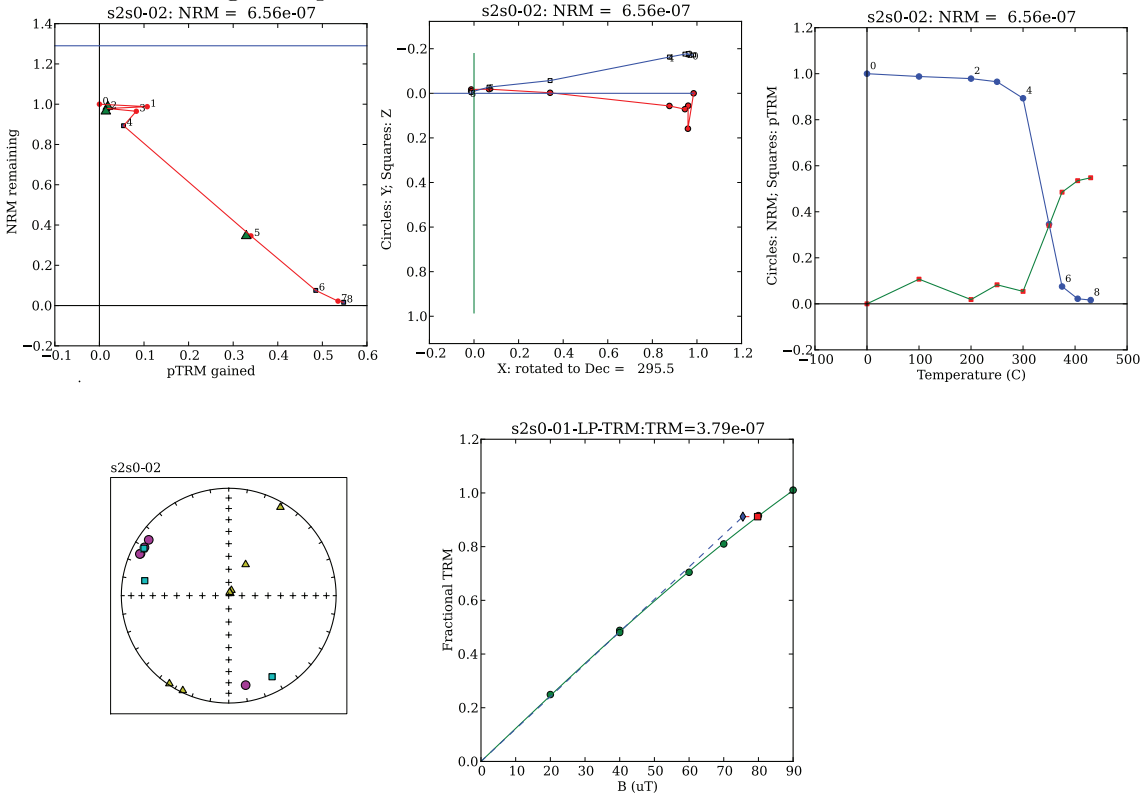
```
% download_magic.py -f zmab0100159tmp01.txt
working on:  'er_locations'
er_locations  data put in  ./er_locations.txt
```

```
working on:  'er_sites'
er_sites  data put in  ./er_sites.txt
working on:  'er_samples'
er_samples  data put in  ./er_samples.txt
....
%mk_redo.py
% thellier_magic_redo.py
Processing  112  specimens - please wait
uncorrected thellier data saved in:  ./thellier_specimens.txt

% thellier_magic.py
.....
16     550   323.6    -4.9 4.370e-08     4.2
17     560   322.1    -2.5 3.240e-08     4.9
Looking up saved interpretation....
Saved interpretation:
specimen     Tmin  Tmax  N  lab_field  B_anc  b  q  f(coe)  Fvds  beta  MAD  Dang

s1p1-01    0  560 18 75.0 79.4 -1.059  54.2 0.919 0.897 0.015      2.0     1.5

pTRM direction=    210.8     85.0  MAD:     3.7
Saved interpretation:
specimen     Tmin  Tmax  N  lab_field  B_anc  b  q  f(coe)  Fvds  beta  MAD  Dang

s1p1-01    0  560 18 75.0 79.4 -1.059  54.2 0.919 0.897 0.015      2.0     1.5

pTRM direction=    210.8     85.0  MAD:     3.7

      s[a]ve plot, set [b]ounds for calculation, [d]elete current interpretation
       [p]revious, [s]ample, [q]uit:

Return for next specimen
s
Enter desired specimen name (or first part there of): s2s0-01
s2s0-01 214 of  269
index step Dec   Inc  Int       Gamma
0     0   194.5   -25.2 3.380e-07
1     100   194.1   -25.2 3.360e-07     0.0
2     200   192.1   -25.4 3.330e-07    11.5
3     250   194.8   -25.5 3.290e-07    76.7
```

```
4     300   193.6   -25.3 3.070e-07    11.7
5     350   191.6   -26.7 1.460e-07     8.1
6     375   188.4   -33.6 3.420e-08     5.5
7     405    36.9   -27.6 6.800e-09     6.1
8     430    29.8   -12.9 2.930e-09     6.2
Looking up saved interpretation....
    None found :(

               s[a]ve plot, set [b]ounds for calculation, [d]elete current interpretation

Return for next specimen
b

Enter index of first point for calculation:  [ 0 ]
return to keep default  0
Enter index  of last point for calculation:  [ 8 ]
return to keep default  8
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 82
         Function evaluations: 157
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 46
         Function evaluations: 89
Non-linear TRM corrected intensity=  79.7699845088
Save this interpretation? [y]/n
y
s2s0-02 215 of  269
index step Dec   Inc  Int       Gamma
0     0   295.5    -9.9 6.560e-07
1     100   304.9   -10.1 6.480e-07    88.7
2     200   298.8   -10.5 6.420e-07    84.8
3     250   299.8   -10.5 6.330e-07    87.9
4     300   299.2   -10.5 5.860e-07    27.0
5     350   295.1    -9.5 2.270e-07     4.5
6     375   280.1   -21.7 4.940e-08     2.0
7     405   169.3   -16.3 1.450e-08     2.1
8     430   151.9   -15.5 1.070e-08     2.6
Looking up saved interpretation....
```

```
    None found :(

                s[a]ve plot, set [b]ounds for calculation, [d]elete current interp

Return for next specimen
q
```

You should have gotten plots that look like these:



When you exit the program, you will have modified the *thellier_specimens.txt* file to include the new interpretation.

5.2.98 thellier_magic_redo.py

[Essentials Chapter 10] and [MagIC] [thellier_magic_redo docs]

This program allows the recalculation of paleointensity experimental data derived from Thellier type experiments. It allows correction for remanence anisotropy from AARM or ATRM ellipsoids (stored in *rmag_anisotropy* format files (see [Essentials Chapter 13] and the aarm_magic.py and atrm_magic.py)

and non-linear TRM corrections, if TRM aquisition data are available (see also trmaq_magic.py).

Apply the corrections to the data you re-interpreted in the thellier_magic.py example, and create a new *pmag_specimens.txt* formatted file, use the program **thellier_magic_redo.py**. First, create a "redo" file using , then re-do the calculations using **thellier_magic_redo.py**, including anisotropy and non-linear TRM corrections. Combine all the specimen files into one using combine_magic.py.

```
% mk_redo.py -f thellier_specimens.txt
%% thellier_magic_redo.py -ANI -NLT
Processing  112  specimens - please wait
WARNING: NO FTEST ON ANISOTROPY PERFORMED BECAUSE OF MISSING SIGMA - DOING CORRECTION ANY
....
anisotropy corrected data saved in:  ./AC_specimens.txt
% combine_magic.py -F pmag_specimens.txt -f thellier_specimens.txt AC_specimens.txt NLT_s
File  ./thellier_specimens.txt  read in with  112  records
File  ./AC_specimens.txt  read in with  112  records
non-linear TRM corrected data saved in:  ./NLT_specimens.txt
All records stored in  ./pmag_specimens.txt
```

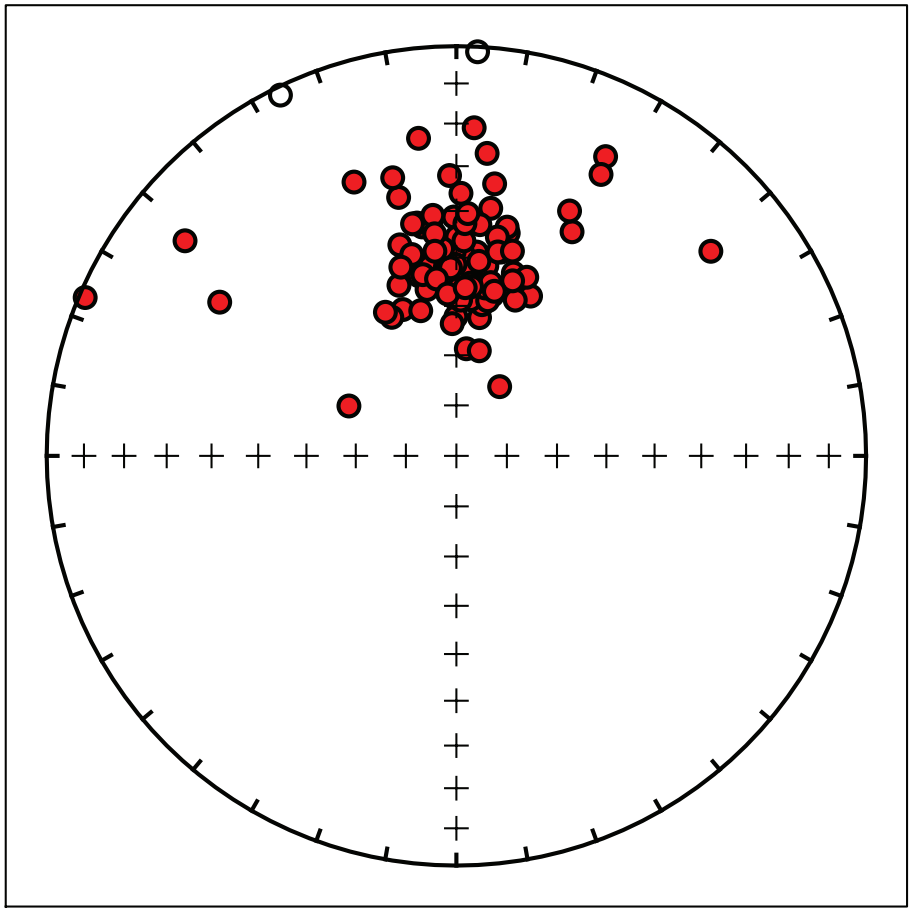
5.2.99 tk03.py

[Essentials Chapter 16] [tk03 docs]

Sometimes it is useful to generate a distribution of synthetic geomagnetic field vectors that you might expect to find from paleosecular variation of the geomagnetic field. The program **tk03.py** generates distributions of field vectors from the PSV model of Tauxe and Kent (2004). Use this program to generate a set of vectors for a latitude of 30° and plot them with eqarea.py.

```
% tk03.py -lat 30 >tk03.out
% eqarea.py -f tk03.out
```

which should give you a plot something like this:



5.2.100 uniform.py

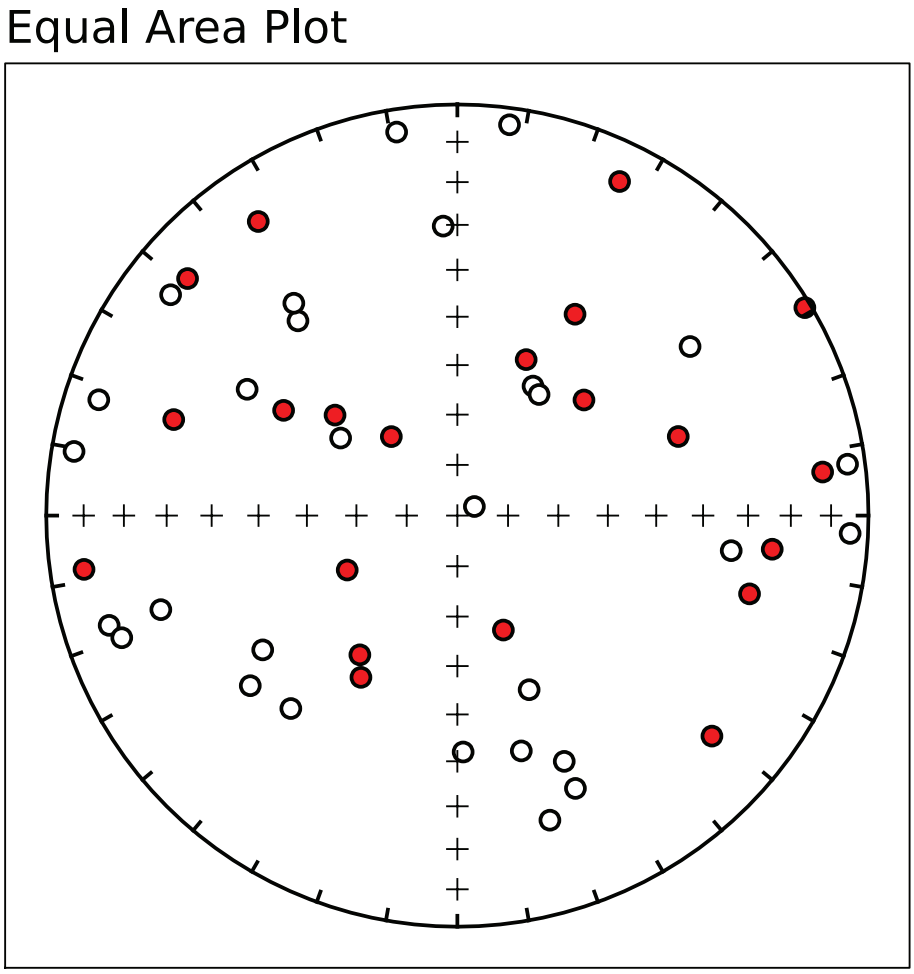
[uniform docs]

It is at times handy to be able to generate a uniformly distributed set of directions (or geographic locations). This is done using a technique described by Fisher et al. (1987).

Draw 50 directions from a uniform distribution and save to a file called *uniform.out*. Then plot the directions with eqarea.py:

```
%  uniform.py -F uniform.out -n 50
% eqarea.py -f uniform.out
```

which should yield a plot something (but not exactly) like this:



5.2.101 update_measurements.py

[MagIC] [update_measurements docs]

In the SIO laboratory, our default specimen naming scheme has a logical relationship to the sample and site names. Sometimes there is no simple relationship and the MagIC import procedure can't parse the sample name into a site name, or the original concept of a site gets changed, for example if several lava flows turn out to be the same one. In any case, there must be a way to change the way data get averaged after the measurement data get converted to a magic_measurments formatted file. To do this, the sample, site relationship can be established in the *er_sample.txt* file for example via the procedure described in orientation_magic.py docuemen-

tation. After the correct relationships are in the *er_samples.txt* file, these must be propagated throughout the rest of the MagIC tables, starting with the *magic_measurements.txt* file. The program **update_measurements.py** will do this at the measurements level. specimens_results_magic.py will then propagate the changes through to the *pmag_results.txt* file.

5.2.102 upload_magic.py

[MagIC] [upload_magic docs]

This program takes all the MagIC formatted files and puts them into a file which can be imported into the MagIC console software for uploading into the MagIC database. As an example, we can "repackage" the file used file downloaded in the download_magic.py example. You could re-interpret that data or fix records with errors, then re-upload the corrected file.

In the *upload_magic* directory in the example datafiles directory, someone has entered all the site latitudes as longitudes and vice versa (surprisingly common!). You can fix this in the *er_sites.txt* file, by swapping the headers sample_lat/sample_lon in the *er_samples.txt* file and site_lat/site_lon in the *er_sites.txt* file then run specimens_results_magic.py to propogate the changes through to the results table. Be sure to set the criteria switch (-exc), the latitude switch (-lat), the geographic coordinate switch (-crd g) and the age switch (-age 0 5 Ma). Then re-package the files for uploading.

AFTER fixing the *er_samples.txt* and *er_sites.txt* files, do the following:

```
% specimens_results_magic.py -exc -lat -age 0 5 Ma -crd g
Acceptance criteria read in from  pmag_criteria.txt

 Pmag Criteria stored in  pmag_criteria.txt

 sites written to  pmag_sites.txt
 results written to  pmag_results.txt


% upload_magic.py
Removing:  ['citation_label', 'compilation', 'calculation_type', 'average_n_lines
 'average_n_planes', 'specimen_grade', 'site_vgp_lat', 'site_vgp_lon', 'direction
 'specimen_Z', 'magic_instrument_codes', 'cooling_rate_corr', 'cooling_rate_mcd']
./er_expeditions.txt is bad or non-existent - skipping
file  ./er_locations.txt  successfully read in
er_locations written to  ./upload.txt
```

```
file  ./er_samples.txt  successfully read in
only first orientation record from er_samples.txt read in
er_samples written to  ./upload.txt
file  ./er_specimens.txt  successfully read in
only measurements that are used for interpretations
.....
now converting to dos file 'upload_dos.txt'
Finished preparing upload file
```

5.2.103 vdm_b.py

[Essentials Chapter 2] [vdm_b docs]

Use this program to try to recover the original field intensity b from a Virtual Dipole Moment of 71.59 ZAm^2 for the latitude 22°.

```
% cat>vdm_b_example.dat
71.59e21 22
^D
% vdm_b.py -f vdm_b_example.dat
 3.300e-05
```

Compare this answer with the original in b_vdm.py, noting that the original input was in millitesla, while this is in tesla.

5.2.104 vector_mean.py

[Essentials Chapter 2] [vector_mean docs]

Create a set of vector data with tk03.py for a latitude of 30°N. Calculate the vector mean of these data.

```
% tk03.py -lat 30 >vector_mean_example.dat
% vector_mean.py -f vector_mean_example.dat
    1.3    49.6    2.289e+06 100
```

5.2.105 vgp_di.py

[Essentials Chapter 2] [vgp_di docs]

Use the program **vgp_di.py** to convert the following:

λ_p	ϕ_p	λ_s	ϕ_s
68	191	33	243

Put the data into a file *vgp_di_example.dat* for example using **cat** on a *NIX operating system. Here is a transcript of one way to use the program which spits out declination, inclination:

```
% vgp_di.py -f vgp_di_example.dat
  335.6    62.9
```

Just for good measure, you could try it out on the data in the di_vgp.py example. See if you get the right answer!

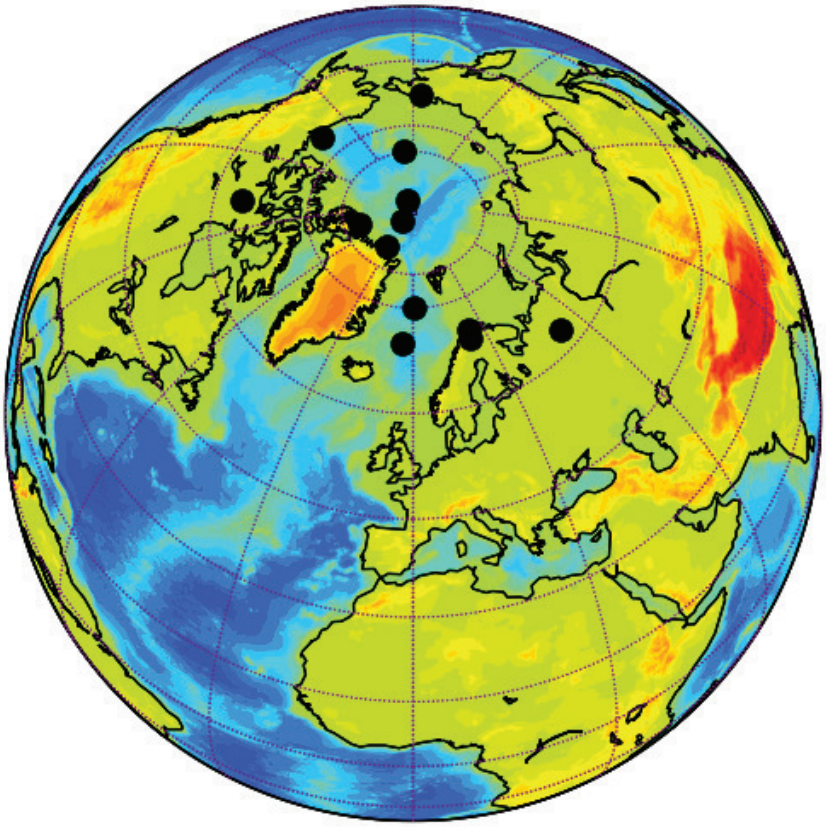
5.2.106 vgpmap_magic.py

[Essentials Chapter 2], [MagIC] and high resolution instructions [vgpmap_magic docs]

Make a plot of the VGPs calculated for the dataset downloaded in the download_magic.py example. Use the high resolution topography (ETOPO 20) dataset and plot the data on an orthographic projection with the viewpoint at 60°N and the Greenwich Meridian (longitude = 0). Make the points black dots with a size of 10pts. Save the file in png format

```
 vgpmap_magic.py -crd g -prj ortho -eye 60 0 -etp -sym ko 10 -fmt png
 S[a]ve to save plot, Return to quit:  a
1  saved in  VGP_map.png
```

You should have a plot like this one:



To use this program, you need to have basemap, which is not part of the free EPD software distribution. Also, if you don't have the high resolution, just leave the -etp switch off to get a plain plot.

5.2.107 watsonsF.py

[Essentials Chapter 11] [watsonsF docs]

First generate two data files with **fishrot.py** with $\kappa = 15$, $N = 10$ and $I = 42$, with $D = 10.$ for the first and 20 for the second:

```
% fishrot.py -k 15 -n 10 -I 42 -D 10 >watsonsF_example_file1.dat
% fishrot.py -k 15 -n 10 -I 42 -D 20 > watsonsF_example_file2.dat
```

To compare these two files using **watsonsF.py**:

```
% watsonsF.py -f watsonsF_example_file1.dat -f2 watsonsF_example_file2.dat
7.74186876032 3.2594 3.2594
```

The first number is Watson's F statistic for these two files (see Essentials Chapter 11) and the second is the number to beat for the two files to be drawn from the same fisher distribution (share a common mean). In this case the data fail this test (F is greater than the required number). Your results may vary!

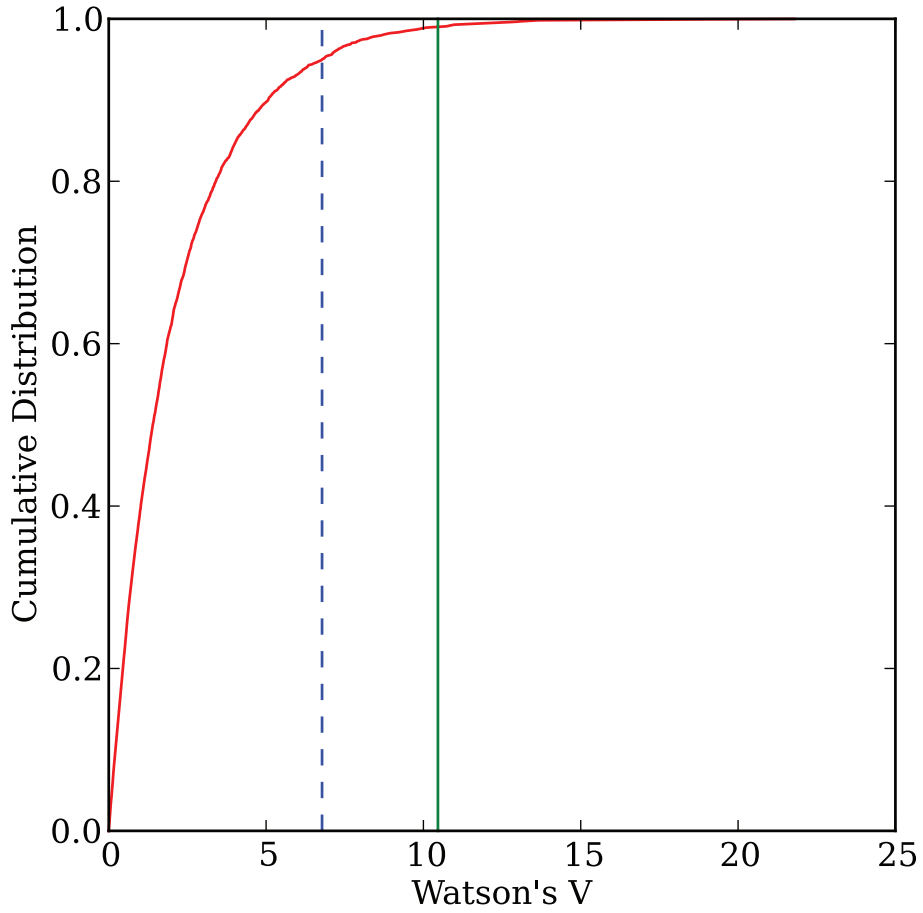
5.2.108 watsonsV.py

[Essentials Chapter 11] [watsonsV docs]

Use the two data files generated in the example for **watsonsF.py** and repeat the test using Watson's V_w statistic.

```
% watsonsV.py -f watsonsF_example_file1.dat -f2 watsonsF_example_file2.dat
Doing  5000  simulations
50
100
150
200
...
Watson's V,  Vcrit:
         10.5        6.8
 S[a]ve to save plot, [q]uit without saving:  a
1  saved in  WatsonsV_watsonsF_example_file1.dat_watsonsF_example_file2.dat.svg
```

which generates the plot:



The two files are significantly different because Watson's V (10.5 in this example) is greater than the V_{crit} value estimated using Monte Carlo simulation (6.8).

Note that your results may vary in detail because every instance of **fishrot.py** generates a different randomly drawn dataset.

5.2.109 zeq.py

[Essentials Chapter 9] [zeq docs]

Use the program **zeq.py** to 1) plot a Zijderveld diagram of the data in *zeq_example.txt*. b) Calculate a best-fit line from 15 to 90 mT. c) Rotate the data such that the best-fit line is projected onto the horizontal axis (instead of the default, North). d) Calcualte a best-fit plane from 5 to 80 mT. Save these plots.

```
% zeq.py -f zeq_example.dat -u mT
```

By selecting 'b', you can pick the bounds and choose 'l' for best-fit line or 'p' for best-fit plane. You can rotate the X-Y axes by selecting 'h' and setting the X axis to 312. Finally, you can save your plots with the 'a' option. You should have saved something like these plots:

The equal area projection has the X direction (usually North in geographic coordinates) to the top. The red line is the X axis of the Zijderveld diagram. Solid symbols are lower hemisphere. The solid (open) symbols in the Zijderveld diagram are X,Y (X,Z) pairs. The demagnetization diagram plots the fractional remanence remaining after each step. The green line is the fraction of the total remanence removed between each step.

5.2.110 zeq_magic.py

[Essentials Chapter 9] & [MagIC] [zeq_magic docs]

Plot the AF demagnetization data available in the file you got in the **download_magic.py** example using **zeq_magic.py**. Use geographic coordinates, where orientations are available.

```
% zeq_magic.py -crd g
sr01a1 0 out of  177
    looking up previous interpretations...
Specimen  N    MAD    DANG  start     end      dec     inc  type  component
sr01a1 9     1.8     1.2   200.0   550.0   331.0    64.4  DE-BFL  A

g: 0      0.0  C  4.065e-05   324.1    66.0
g: 1    100.0  C  3.943e-05   330.5    64.6
g: 2    150.0  C  3.908e-05   324.9    65.5
g: 3    200.0  C  3.867e-05   329.4    64.6
g: 4    250.0  C  3.797e-05   330.3    64.5
g: 5    300.0  C  3.627e-05   330.1    64.0
g: 6    350.0  C  3.398e-05   327.0    64.4
g: 7    400.0  C  2.876e-05   328.2    64.0
g: 8    450.0  C  2.148e-05   323.8    65.2
g: 9    500.0  C  1.704e-05   326.0    63.9
g: 10    525.0  C  1.200e-05   326.5    63.7
g: 11    550.0  C  5.619e-06   325.5    64.4

 g/b: indicates  good/bad measurement.  "bad" measurements excluded from calculat

  set s[a]ve plot, [b]ounds for pca and calculate, [p]revious, [s]pecimen,
   change [h]orizontal projection angle,   change [c]oordinate systems,
 [d]elete current interpretation(s), [e]dit data,   [q]uit:

<Return>  for  next specimen
```

The plots will look similar to the zeq.py example, but the default here is for the X-axis to be rotated to the specimen's NRM direction (note how the X direction is rotated off from the top of the equal area projection).

5.2.111 zeq_magic_redo.py

[Essentials Chapter 9] & [MagIC] [zeq_magic_redo docs]

In the *zeq_magic* directory, use the program **zeq_magic_redo.py** to create a *pmag_specimens* formatted file with data in geographic coordinates from the sample coordinate file *zeq_specimens.txt*. Assuming that sample orientations are in a file called *er_samples.txt*, use mk_redo.py first to create file called *zeq_redo*. Then use **zeq_magic_redo.py** to create two *pmag_specimen* formatted files: one in specimen coordinates *zeq_specimens_s.txt* and one in geographic coordinates *zeq_specimens_g.txt*. Combine these into one file called *pmag_specimens.txt*.

Note that indented lines belong with the line above as a single line.

```
%mk_redo.py
% zeq_magic_redo.py -F zeq_specimens_s.txt -crd s
Processing  175  specimens - please wait
Recalculated specimen data stored in  ./zeq_specimens_s.txt

% zeq_magic_redo.py -F zeq_specimens_g.txt -crd g
Processing  175  specimens - please wait
Recalculated specimen data stored in  ./zeq_specimens_g.txt
% combine_magic.py -F pmag_specimens.txt   -f zeq_specimens_s.txt zeq_specimens_g.txt
```

5.3 Complaints department

Most of the software and this documentation were written by Lisa Tauxe (ltauxe@usd.edu) who is solely responsible for all bugs and boo-boos. Please contact me with bug reports, suggestions and requests.

Chapter 6

Introduction to Python Programming

Type 'python' on your command line.

You have just fired up Python and you are in an interactive mode with the prompt >>>. Now, you can just start typing commands. After each command, press the return key and see what happens:

```
>>> a=2
>>> a
2
>>> b=2
>>> c=a+b
>>> c
>>> c+=1
>>> c
5
>>> a=2; b=2; c=a+b; c
4
>>> d,e,f=4,5,6 # note syntax! d=4;e=5;f=6
>>> # note also that the symbol '#' means the rest of the line is a comment!
```

To get out of Python interactive mode and back to your beloved command line, type the control key (here-after ^) and D at the same time. From your school experience of Algebra, you will recognize a, b, and c in the above session as *variables* and + as an *operation*. If you have previous programming experience, you may have been surprised that we didn't

declare variables up front (C programmers always have to). And, the variables above are obviously behaving as integers, not floating point variables (no decimals) but they are not letters between i and n. And there were funny lines that seemed to set three numbers at once:

```
>>> a=2; b=2; c=a+b; c
and
>>> d,e,f=4,5,6 # note syntax! d=4;e=5;f=6
```

Here are some rules governing variables and operations in Python:

- Variable names are composed of alphanumeric characters, including '-' and '_'.
- They are case sensitive: 'a' is not the same as 'A'.
- They do NOT have to be specified in advance (unlike C)
- In fortran, integers are $i - n$ and floating points are all else - not the case in Python. You can make them whatever you want.
- + adds, - subtracts, * multiplies, / divides, % gives the remainder, ** raises to the power
- These two are fun: += and -=. They add to and subtract from respectively.
- Parentheses determine order of operation (as in any reasonable programming language).
- For math functions, we can use various modules that either come standard with python (the math module) or are additions that come with the Enthought Python Edition we are using (the NumPy module). A module is just a collection of functions. NB: There is online help for any python function or method: just type help(FUNC).

6.1 A first look at NumPy

O.K. First of all - how do you pronounce 'NumPy'. According to Important People at Enthought (e.g, Robert Kern), it should be pronounced "Num" as in "Number" and "Pie" as in, well, pie, or Python. Oops. It is way more fun to say Numpee!

So, that out of the way, what can NumPy do for us? Turns out, a whole heck of a lot! But for now we will just scratch the surface. It can, for example, give us the value of π as numpy.pi. Note how the module name comes first, then the name of the function we wish to use. In this case, the function just returns the value of π.

To use NumPy functions, we must first "import" the module with the command import. The first time you do this after installing Python, it may take a while, but after that it should be very quick.

There are four styles of the import command which all do pretty much the same thing but differ in how you have to call the function after importing:

```
>>>  import numpy
>>> numpy.pi
3.1415926535897931
```

This makes all the functions in NumPy available to you, but you have to call them with the numpy.FUNC syntax.

```
>>>  import numpy as np  # or anything else!  e.g., some use:  N
>>> np.pi  # or N.pi in the second case.
3.1415926535897931
```

This does the same as the first, but allows you to call NumPy anything you want - to save typing? To import all the functions from NumPy and not have to type numpy at all:

```
>>> from numpy  import *
>>> pi
3.1415926535897931
```

This imports all the umpty-ump functions, which is a heavy load on your memory, but you can also just import a few, like pi or sqrt:

```
>>> from numpy import pi, sqrt # pi and square root
>>> pi
3.1415926535897931
>>> sqrt(4)
2.0
```

Did you notice how sqrt(4) where 4 was an integer, returned a floating point variable (2.0)?

Good housekeeping Tip #1: I tend to import modules using the first option above. That way I know what module the functions I'm using are coming from - especially because we don't know off-hand ALL the functions available in any given module and there might be conflicts with my own function names or two different modules could have the same function (like math and numpy).

Here is a (partial) list of some useful NumPy functions:

absolute(x)	absolute value
arccos(x)	arccosine
arcsin(x)	arcsine
arctan(x)	arctangent
arctan2(y,x)	arctangent of y/x in correct quadrant (***very useful!)
cos(x)	cosine
cosh(x)	hyperbolic cosine
exp(x)	exponential
log(x)	natural logarithm
log10(x)	base 10 log
sin(x)	sine
sinh(x)	hyperbolic sine
sqrt(x)	square root
tan(x)	tangent
tanh(x)	hyperbolic tangent

Note that in the trigonometric functions, the argument is in RADIANS!.You can convert from degrees to radians by multiplying by: numpy.pi/180.. Also notice how these functions have parentheses, as opposed to numpy.pi which has none. The difference is that these take arguments, while numpy.pi just returns the value of π.

If you are desperate, you can always use your Python interpreter as a calculator:

```
>>> import numpy
>>> a=2
>>> b=-12
>>> c=16
>>> quad1 = (-b + numpy.sqrt(b**2-4.*a*c))/(2.*a)
>>> quad1
4
>>> y=numpy.sin(numpy.pi/6.)
>>> y
0.5
```

6.2 Variable types

The time has come to talk about variable types. We've been very relaxed up to now, because we don't have to declare them up front and we can often even change them from one type to another on the fly. But - variable types matter, so here goes. Python has integer, floating point (both long and short), string and complex variable types. It is pretty clever about figuring out what is required. Here are some examples:

```
>>> number=1 # an integer
>>> Number=1.0 # a floating point
>>> NUMBER='1' # a string
>>> complex=1j # a complex number with imaginary part 1
>>> complex(3,1) # the complex number 3+1i
```

Try doing math with these!

```
>>> number+number
2     [ an integer]
>>> number+Number
2.0   [ a float]
>>> NUMBER+NUMBER
11 [a string]
>>> number+NUMBER  [Gives you an angry error message]
```

Lesson learned: you can't add a number and a string. and string addition is different! But you really have to be careful with this. If you multiply a float by an integer, it is possible that you will convert the float to an integer when you really wanted all those numbers after the decimal! So, if you want a float, use a float.

You can convert from one type to another (if appropriate) with:

```
int(Number); str(number);  float(NUMBER);
long(Number); complex(real,imag)
```

long() converts to a double precision floating point and complex() converts the two parts to a complex number.

There is another kind of variable called "boolean". These are: true, false, and, or, and not. For the record, the integer '1' is true and '0' is false. These can be used to control the flow of the program as we shall learn later.

6.3 Data Structures

In previous programming experience, you may have encountered arrays, which are a nice way to group a sequence of numbers that belong together. In Python we also have arrays, but we also have more flexible data structures, like lists, tuples, and dictionaries, that group arbitrary variables together, like strings and integers and floats - whatever you want really. We'll go through some attributes of the various data structures, starting with lists.

Lists

- Lists are denoted with square brackets, [ ], and can contain any arbitrary set of items, including other lists!
- Items in the list are referred to by an index number, starting with 0.
- You can count from the end to the beginning by starting with -1 (the last item in the list), -2 (second to last), etc.
- Items can be sorted, deleted, inserted, sliced, counted, concatenated, replaced, added on, etc.

Examples:

```
>>> mylist=['a',2.0,'400','spam',42,[24,2]] # defines a list
>>> mylist[2] # refers to the third item
'400'
>>> mylist[-1] # refers to the last item
[24,2]
>>> mylist[1]=26.3   # replaces the second item
>>> del mylist[3] # deletes the fourth element
```

To slice out a chunk of the middle of a list:

```
>>> newlist=mylist[1:3]
```

This takes items 2 and 3 out (note it takes out up to but not including the last item number - don't ask me why). Or, we can slice it this way:

```
>>> newlist=mylist[3:]
```

which takes from the fourth item (starting from 0!) to the end.

To copy a list BEWARE! You can make a copy - but it isn't an independent copy (like in, e.g., Fortran), but it is just another name for the SAME OBJECT, so:

```
>>> mycopy=mylist
>>>mylist[2]='new'
>>>mycopy[2]
'new'
```

See how mycopy got changed when we changed mylist? To spawn a new list that is a copy, but an independent entity:

```
>>>mycopy=mylist[:]
```

Now try:

```
>>>mylist[2]=1003
>>>mycopy[2]
'new'
```

So now mycopy stayed the way it was, even as mylist changed.

Python is "object oriented", a popular concept in coding circles. We'll learn more about what that means later, but for right now you can walk around feeling smug that you are learning an object oriented programming language. O.K., what is an object? Well, mylist is an object. Cool. What do objects have that might be handy? Objects have "methods" which allow you to do things to them. Methods have the form: object.method()
Here are two examples:

```
>>> mylist.append('me too') # appends a string to mylist
>>> mylist.sort() # sorts the list alphanumerically
>>> mylist
[2.0, 42, [24, 2], '400', 'a', 'me too', 'spam']
```

For a complete list of methods for lists, see: `http://docs.python.org/tutorial/datastructures.html#more-on-lists`

More about strings

Numbers are numbers. While there are more kinds of numbers (complex, etc.), strings can be more interesting. Unlike in some languages, they can be denoted with single, double or triple quotes: e.g., 'spam', "Sam's spam", or

```
"""
Hi there I can type as
many lines as I want
"""
```

Strings can be added together (newstring = 'spam' + 'alot'). They can be sliced (newerstring = newstring[0:3]). but they CANNOT be changed in place - you can't do this: newstring[0]='b'. To find more of the things you can and cannot do to strings, see:

`http://docs.python.org/tutorial/introduction.html#strings`

Data structures as objects

Tuples

What? Tuples? Tuples consist of a number of values separated by commas. They are denoted with parentheses.

```
>>> t = 1234, 2.0, 'hello'
>>> t
(1234, 2.0, 'hello')
>>> t[0]
1234
```

Tuples are sort of like lists, but like strings, their elements cannot be changed. However, you can slice, concatenate, etc. For more see:

`http://docs.python.org/tutorial/datastructures.html#tuples-and-sequences`

Dictionaries!

Dictionaries are denoted by {}. They are also somewhat like lists, but instead of integer indices, they use alphanumeric 'keys': I love dictionaries. So here is a bit more about them.
To define one:

```
>>> Telnos={'lisa':46084,'lab':46531,'jeff':44707} # defines a dictionary
```

To return the value associated with a specific key:

```
>>> Telnos['lisa']
46084
```

To change a key value:

```
>>> Telnos['lisa']=46048
```

To add a new key value:

```
>>> Telnos['newguy']=48888
```

Dictionaries also have some methods. One useful one is:

```
>>> Telnos.keys()
['lisa','lab','jeff','newguy']
```

which returns a list of all the keys.

For a more complete accounting of dictionaries, see:
`http://docs.python.org/tutorial/datastructures.html#dictionaries`

N-dimensional arrays

Arrays in Python have many similarities to lists. Unlike lists, however, arrays have to be all of the same data type (dtype), usually numbers (integers or floats), although there appears to be something called a character array. Also, the size and shape of an array must be known *a priori* and not determined on the fly like lists. For example we can define a list with L=[], then append to it as desired, but not so arrays - they are much pickier and we'll see how to set them up later.

Why use arrays when you can use lists? They are far more efficient than lists particularly for things like matrix math. But just to make things a little confusing, there are several different data objects that are loosely called arrays, e.g., arrays, character arrays and matrices. These are all subclasses of ndarray. I'm just going to briefly introduce arrays and matrices here.

Here are a few ways of making arrays:

```
>>> import numpy
>>> A= numpy.array([[1,2,3],[4,2,0],[1,1,2]])
>>> A
array([[1, 2, 3],
       [4, 2, 0],
       [1, 1, 2]])
>>> B=numpy.arange(0,10,1).reshape(2,5)
>>> B
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> C=numpy.array([[1,2,3],[4,5,6]],numpy.int32)
>>> C
array([[1, 2, 3],
       [4, 5, 6]])
>>> D=numpy.zeros((2,3)) # Notice the zeros and the size is specified by a tuple.
>>> D
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> E=numpy.ones((2,4))
>>> E
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> F=numpy.linspace(0,10,14)
>>> F
array([  0.        ,   0.76923077,   1.53846154,   2.30769231,
         3.07692308,   3.84615385,   4.61538462,   5.38461538,
         6.15384615,   6.92307692,   7.69230769,   8.46153846,
         9.23076923,  10.        ])
>>> G=numpy.ndarray(shape=(2,2), dtype=float)
>>> G   # note how this is initalized with really low numbers (but not zeros).
array([[  1.90979621e-313,   2.75303490e-308],
       [  1.08539798e-071,   3.05363949e-309]])
```

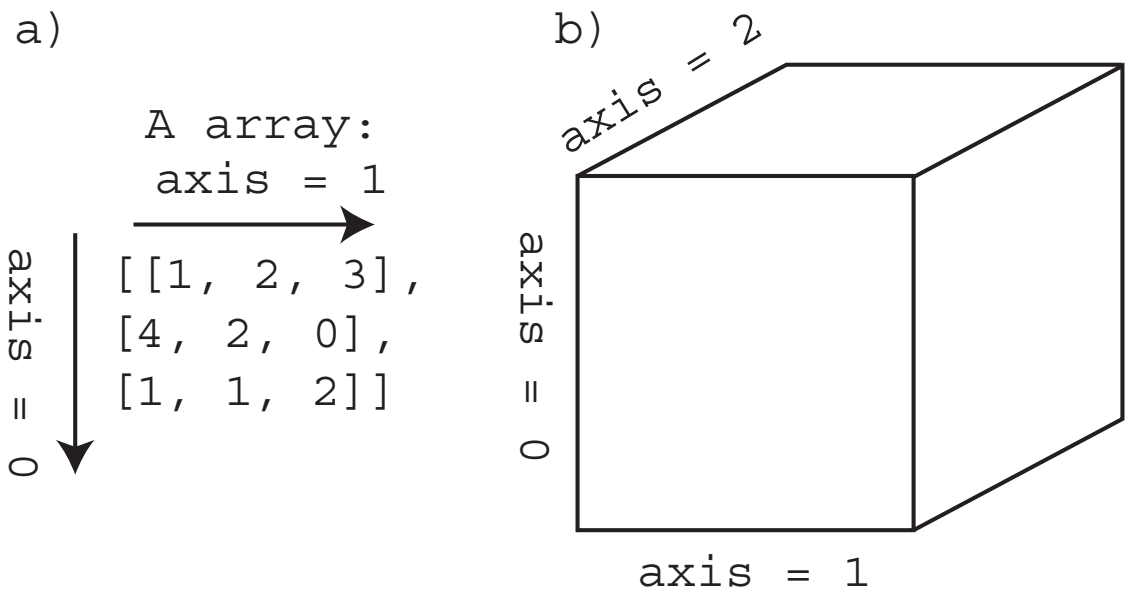
Note the difference between linspace(start,stop,N) and arange(start,stop,step). The function linspace creates an array with 14 evenly spaced elements between the start and stop values while arange creates an array with elements at colorbluestep intervals between the starting and stopping values. In some of the online examples you will find the short-cuts for arange() and linspace as r_(-5,5,20j) and r_(-5,5,1.) respectively.

Python arrays have methods like dtype, ndim, shape, size, reshape(), ravel(), transpose() etc. Did you notice how some of these require parentheses and some don't? The answer is that some of these are functions and some are classes, both of which we will get to later.

Let's see what the methods can do. First, arrays made in the above example are of different data types. To find out what data type an array is, just use the method dtype as in:

```
>>> D.dtype
dtype('float64')
>>>
```

And of course arrays, unlike lists have dimensions and shape. Dimensions tell us how many axes there are with axes defined as in this illustration:



As shown above our A array has two dimensions (axis 0 and 1). To get Python to tell us this, we use the ndim method:

```
>>>  A= numpy.array([[1,2,3],[4,2,0],[1,1,2]]) # just to remind you
>>> A.ndim
2
```

Notice how zeros, ones and ndarray used a shape tuple in order to define the arrays in the examples above. The shape of an array is how many elements are along each axis. So, naturally we see that the C array is a 2x3 array. Python returns a tuple with the shape information using the shape method:

```
>>> C.shape
(2, 3)
```

Let's say we don't want a 2x3 array for the sequence in the array C, but we want a 3x2 array. Python can reshape an array with a different shape tuple like this:

```
>>> C.reshape((3,2))
array([[1, 2],
       [3, 4],
       [5, 6]])
```

And sometimes we just want all the elements lined up along one axis. We could do that with reshape of course using a tuple with the size of the array (the total number of elements). You can see that this is 6 here. We could even get python to tell us what the size is (C.size) and use that in the reshape size tuple. Alternatively we can use the ravel() method which doesn't require us to know the size in advance:

```
>>> C.ravel()
array([1, 2, 3, 4, 5, 6])
```

There are other ways to reshape, slice and dice arrays. The syntax for slicing of arrays is similar to that for lists:

```
>>> B=A[0:2] # carve the top two lines off of matrix A from above
array([[1, 2, 3],
       [4, 5, 6]])
```

Lots of applications in Earth Science require the transpose of an array:

```
>>> A.transpose() # this is the same as A.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Also, we can concatenate two arrays together with the - you guessed it - concatenate() method. For a lot more tricks with arrays, go to the NumPy Reference website here: `http://docs.scipy.org/doc/numpy/reference/`.

To convert the A array to a list: L=A.tolist(), from a list or tuple to an array: A=numpy.array(L), or from a list, a tuple or an array to a NumPy array: a=numpy.asarray(L)

Let's go a bit deeper into slicing of arrays. First a review of lists. You will recall that in python, indexing starts with 0, so for the list L=[0,2,4,6,8], L[1] is 2. The index of the last item is -1, so L[-1]=8. To find out what the index for the number 4 is, for example, we have the index() method: L.index(4), which will return the number 2. We actually already used this method when we implemented command line arguments, but it wasn't really explained. We know that to reassign a given index a new value we use the syntax L[1]=2.5. And to use a part of a list (a slice) we use, e.g., B=L[2:4], which defines B as a list with L's elements 2 and 3 (4 and 6). And you also know that B=L[2:] takes all the elements from 2 to the end. From these

examples, you can infer that the basic syntax for slicing is [start:stop:step]; if the step is omitted it is assumed to be 1.

Arrays (and matrices) work in a similar fashion to lists, but these are multidimensional objects, so things get hairy fast. The basic syntax is the same: [start:stop:step], or i:j:k. but with Python arrays, we step through all the j's for each i at step k. This is best shown with examples:

```
>>> import numpy
>>> A=numpy.linspace(0,29,30)
>>> B=A.reshape(5,6)
array([[  0.,   1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.,  16.,  17.],
       [ 18.,  19.,  20.,  21.,  22.,  23.],
       [ 24.,  25.,  26.,  27.,  28.,  29.]])
>>> B[1:3,:-1:2]
array([[  6.,   8.,  10.],
       [ 12.,  14.,  16.]])
```

Let's pick about the statement B[1:3,:-1:2] to see if we can understand what it does. The first part alone returns lines 2 and 3:

```
>>> B[1:3]
array([[  6.,   7.,   8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.,  16.,  17.]])
```

Here j goes from [:-1], in other words, we all but the last element:

```
>>> B[1:3,:-1]
array([[  6.,   7.,   8.,   9.,  10.],
       [ 12.,  13.,  14.,  15.,  16.]])
```

And finally, we have the step of 2, which takes every other element:

```
>>> B[1:3,:-1:2]
array([[  6.,   8.,  10.],
       [ 12.,  14.,  16.]])
```

For more on array slicing (indexing), see:

`http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html`

6.4 Python Scripts

Are you tired of typing yet? Python scripts are programs that can be run and re-run from the command line. You can type in the same stuff you've been doing interactively into a script file (ending in .py). You can edit scripts with your favorite text editor (NOT Word!). And then you can run them like this:

```
%python < myscript.py
```

On a Mac or Unix system, you can put in a header line identifying the script as python (#!/usr/bin/env python), make it executable (chmod a+x) and run it like this:

```
% myscript.py
```

On PCs, you can just type the name of the script without the header or the chmod command. But, because PCs don't come standard with a **cat** command, you have to create the file with Notepad, instead of with **cat** for all the examples using **cat**.

Here is an example that creates a script using the Unix **cat** command, makes it executable and then runs it:

```
% cat > printmess.py
#!/usr/bin/env python
# simple Python test program (printmess.py)
print 'test message'
^-D
% chmod a+x printmess.py
% ./printmess.py
test message
```

In a Python script on Unix machines (including MacOS), the first line MUST be:

```
#! /usr/bin/env python
```

so that the file is interpreted as Python. Unlike Fortran or C, you CANNOT start with a comment line (try switching lines 1 and 2 and see what happens).

The second line is a comment line. Anything to the right of # is assumed to be a comment. Notice that print goes by default to your screen. Because

the message is a string, you can use single or double quotes for the test message. You can get an apostrophe in your output by using double quotes and quote marks by using single quotes, i.e.,

```
#! /usr/bin/env python
# simple Python test program 2 (printmess2.py)
print "The pump don't work 'cuz the vandals took the handles"
print 'She said "I know what it\'s like to be dead"'
```

produces:

```
% ./printmess2.py
The pump don't work 'cuz the vandals took the handles
She said "I know what it's like to be dead"
%
```

In the second print statement, the \' is necessary to prevent an error (try it). This is an example of a Python 'escape code'. These are used to escape some special meaning, as in an end-quote for a string in this example. We use the backslash to say that we really really want a quote mark here. Other escape codes are listed here:

`http://www.python-course.eu/variables.php`

Here's another example of a program - this one has an typo in line 4:

```
#! /usr/bin/env python
abeg = 2.1
aend = 3.9
adif = aend - abge
print 'adif = ', adif
```

You had intended to type 'abeg' but typed 'abge' instead. When you run the program, you get an error message:

```
Traceback (most recent call last):
  File "./undeclared.py", line 4, in <module>
    adif = aend - abge
NameError: name 'abge' is not defined
```

Error messages are a desirable feature of Python. You don't want the program to run by assigning some arbitrary value to abge and giving you a wrong answer. Yet many languages will do exactly that.

6.5 Code blocks

Any reasonable programming language must provide a way to group blocks of code together, to be executed under certain conditions. In Fortran, for example, there are if statements and do loops which are bounded by the statements if, endif and do, endo respectively. Many of these programs encourage the use of indentation to make the code more readable, but do not require it. In Python, indentation is the way that code blocks are defined - there are no terminating statements. Also, the initiating statement terminates in a colon. The trick is that all code indented the same number of spaces (or tabs) to the right belong together. The code block terminates when the next line is less indented. A typical Python program looks like this:

```
program statement
block 1 top statement:
    block 1 statement
    block 1 statement \
        ha-ha i can break the indentation convention!
    block 1 statement
    block 2 top statement:
        block 2 statement
        block 2 statement
        block 3 top statement:
            block 3 statement
            block 3 statement
            block 4 top statement: block 4 single line of code
        block 2 statement
        block 2 statement
    block 1 statement
    block 1 statement
program statement
```

Exceptions to the code indentation rules are:

- Any statement can be continued on the next line with the continuation character \ and the indentation of the following line is arbitrary.
- If a code block consists of a single statement, then that may be placed on the same line as the colon.

- The command break breaks you out of the code block. Use with caution!
- There is a cheat that comes in handy when you are writing a complicated program and want to put in the code blocks but don't want them to DO anything yet: the command pass does nothing and can be used to stand in for a code block.

Good housekeeping Tip #2: Always use only spaces or only tabs in your code indentation. I use only spaces because I use vi to write my code. Others use Xcode, the Python IDLE program, or TextWrangler to write their code and some of these things use tabs by default. Whatever you do BE CONSISTENT because tabs are not the same as spaces in Python even if you can't tell the difference just by looking at it.

In the following, I'll show you how Python uses code blocks to create "do" and "while" loops, and "if" statements.

The for loop

Looping through lists

Here is an example of a simple "for loop":

```
#!/usr/bin/env python
mylist=[42,'spam','ocelot']
for i in range(0,len(mylist),1): # note absence of Indices list, start and step
   print mylist[i]
print 'All done'
```

This script creates the list mylist with the line mylist=[42,'spam','ocelot']. The length of mylist is an integer value returned by len(mylist). The script uses this integer as the 'stop' value in the range() function, which returns a list of integers from 0 to the stop value MINUS ONE at intervals of one. [The minus one convention is hard to get use to for Fortran programmers, but it is typical of Python syntax (and also of C) so just deal with it.] Anyway, range(start,stop,step) is just like numpy.arange(start,stop,step) but returns integers instead of floats. Also, like numpy.arange(), there is a short hand form when the minimum is zero and the interval is one, so we could (and will) just use the command range(stop).

Python makes i step through the list of numbers from 0 to 2, printing the i^{th} element of mylist. Note how the print command is indented - this is the program block that is executed for each i. Note also that the line

could have been on the previous line after the colon, because there is only one line in the program block. But never-mind, this way works too. When i finishes it's business, the program block terminates. At that point, the program prints out the 'All done' string. There is no "enddo" statement or equivalent in Python.

But, Python is far more fun than the old-school for i in syntax in the above code snippet. In Python we can just step through a list directly. Here is another script which does just that (why not?):

```
#!/usr/bin/env python
mylist=[42,'spam','ocelot']
for item in mylist: # note absence of range statement
    print item
print 'All done'
```

Note that of course we could have used any variable name instead of 'item', but it makes sense to use variable names that mean what they do. It is easier to understand what 'item' stands for than just the Fortran style of i.

Here is an example with a little more heft to it. It creates a table of trigonometry functions, spitting them out with a formatted print statement:

```
#! /usr/bin/env python
import numpy as np
deg2rad = np.pi/180. # remember conversion to radians
for theta in range(90): # short form of range, returns [0,1,2...89]
   ctheta = np.cos(theta*deg2rad) # define ctheta as cosine of theta
   stheta = np.sin(theta*deg2rad)# define stheta as  sine of theta
   ttheta = np.tan(theta*deg2rad)   # define ttheta as tangent of theta
   print '%5.1f %8.4f %8.4f %8.4f' %(theta, ctheta, stheta, ttheta)
```

Let's pick this one apart a bit. First, notice the use of the variable deg2rad to convert from degrees to radians. Also notice how deg2rad is defined: deg2rad = np.pi/180. using the NumPy function for π and the decimal point after 180. While in this case, it makes absolutely no difference (try it!), it is a good practice to use real numbers if you want your variable to stay real. In fact:

Good housekeeping Tip #3: Always use a decimal if you want your variable to be a floating point variable.

The expression ctheta = np.cos(theta*deg2rad) uses the numpy cosine function. Ideally theta should be a real variable while in fact it is an integer

in this expression, but fortunately Python figures that out and converts it to a real. Note that we could have also converted theta to a float first with the command float(theta).

```
print '%5.1f %8.4f %8.4f %8.4f' %(theta, ctheta, stheta, ttheta)
```

To make the output look nice, we do not use

```
print theta, ctheta, stheta, ttheta
```

which would space the numbers irregularly among the columns and put out really long numbers. Instead, we explicitly specify the output format. The output format is given in the quotes. The format for each number follows the %, 5.1f is for 5 spaces of floating point output, with 1 space to the right of the decimal point. The single blank space between %5.1f and %8.4f is included in the output, in fact any text there is reproduced exactly in the output, thus to put commas between the output numbers, write:

```
print '%5.1f, %8.4f, %8.4f, %8.4f' %(theta, ctheta, stheta, ttheta)
```

Tabs (\t) would be formatted like this:

```
print '%5.1f '\t' %8.4f'\t'  %8.4f,\t' %8.4f' %(theta, ctheta, stheta, ttheta)
```

Looping through arrays

We just learned that for loops with lists just step through item by item. In n-dimensional arrays, they steps through row by row (like in slicing). For example,

```
>>> for r in B:
...     print r
...
[ 0.  1.  2.  3.  4.  5.]
[  6.   7.   8.   9.  10.  11.]
[ 12.  13.  14.  15.  16.  17.]
[ 18.  19.  20.  21.  22.  23.]
[ 24.  25.  26.  27.  28.  29.]
```

If you really want to step through element by element, you can use the ravel() method which flattens an N-dimensional array to a single dimension:

```
>>> for e in B.ravel():
...     print  e
...
0.0
1.0
2.0
3.0
etc.
```

For more on looping (or iterating), see:
`http://docs.scipy.org/doc/numpy/reference/arrays.nditer.html`

If and while blocks

The "for loop" is just one way of controlling flow in Python. There are also if and while code blocks. These execute code blocks the same way as for loops (colon terminated top statements, indented text, etc.). For both of these, the code block is executed if the top statement is TRUE. For the "if" block, the code is executed once but in a "while" block, the code keeps executing as long as the statement remains TRUE.

The key to flow control therefore is in the top statement of each code block; if it is TRUE, then execute, otherwise skip it. To decide if something is TRUE or not (in the boolean sense), we need to evaluate a statement using comparisons. Here's a handy table with comparisons (relational operators) in different languages:

Comparisons

F 77	F90	C	MATLAB	PYTHON	meaning
.eq.	==	==	==	==	equals
.ne.	/=	!=	∼=	!=	does not equal
.lt.	<	¡<	<	<	less than
.le.	<=	<=	<=	<=	less than or equal to
.gt.	>	>	>	>	greater than
.ge.	> =	> =	> =	> =	greater than or equal to
.and.	.and.	&		&	and
.or.	.or.	——	—	or	

These operators can be combined to make complex tests. Here is a juicy complicated statement:

```
if ( (a > b and c <= 0) or d == 0):
    code block
```

There are rules for the order of operations for these things like, multiplication gets done before addition. But these are easy to forget. You can look it up in the documentation if you are unsure or, better, just put in enough parenthesis to make it completely clear to anyone reading your code.
Good housekeeping Tip #4: Use parentheses liberally - make the order of operation completely unambiguous even if you could get away with fewer.

One nice aspect of Python compared to C is that if you make a mistake and type, for example,

```
if (a = 0):
```

you will get an error message during compilation. In C this is a valid statement with a completely different meaning than is intended!

Finer points of 'if' blocks

The simplest 'if' block works just like we have described:

```
if (2+2)==4: # note the use of '==' and parentheses in comparison statement
   print 'I can put two and two together!'
```

However, as in any other reasonable programming language, there are whistles and bells to the 'if' code blocks. In Python these are: elif and else. A code block gets executed if the top if statement is FALSE and the elif statement is TRUE. If both the top if and the elif statements are FALSE but the else statement is TRUE, then Python will execute the block following the else. Consider these examples:

```
#!/usr/bin/env python
mylist=['jane','doug','denise']
if 'susie' in mylist:
    pass # don't do anything
if 'susie' not in mylist:
   print 'call susie and apologize!'
   mylist.append('susie')
elif 'george' in mylist: # if first statement is false, try this one
    print 'susie and george both in list'
else: # if both statements are false, do this:
   print "susie in list but george isn't"
```

While loops

As already mentioned, the 'while' block continues executing as long as the while top statement is TRUE. In other words, the if block is only executed once, while the while block keeps looping until the statement turns FALSE. Here are a few examples:

```
#!/usr/bin/env python
a=1
while a < 10:
    print a
    a+=1
print "I'm done counting!"
```

Code blocks in interactive Python

All of these program blocks can also be done in an interactive session also using indentation. The interactive shell responds with '.....' instead of '>>>' once you type a statement it recognizes as a top statement. To signal that you are done with the program block, simply hit return:

```
>>> a=1
>>> while a<10:
....    print a
....    a+=1
....[return to execute block]
```

6.6 File I/O in Python

Python would be no better than a rather awkward graphing calculator (and we haven't even gotten to the graphing part yet) if we couldn't read data in and spit data out. You learned a rudimentary way of spitting stuff out already using the print statement, but there is a lot more to file I/O in Python. We would like to be able to read in a variety of file formats and output the data any way we want. In the following we will explore some of the more useful I/O options in Python.

Reading data in

From a file

If you are using Python interactively or want interactivity in a script, use the command: raw_input(). It acts as a prompt and reads in whatever is supplied prior to a return as a string.

```
X=[] # make a list to put the data in
ans=float(raw_input("Input numeric value for X:  "))
X.append(ans) # append the value to X
print X[-1] # print the last item in the list
```

In this example, the variable ans will be read in as a string variable, converted to a float and appended to the list, X. raw_input() is a simple but rather annoying way to enter things into a program. Another (less annoying) way is put the data in a file (e.g., myfile.txt) with cat, paste, Excel (saved as a text file), or whatever and read it into Python. The procedure is straight-forward: we must first open the file, then read it in and parse lines into the desired variables.

To open a file we use the command open(), one of Python's built-in functions. For a complete list of these, see:

`http://docs.python.org/library/functions.html`

The open() function returns an object, complete with methods, like readlines() which, yes, reads all the lines. Suppose you have a file containing the coordinates of some seismic stations which you want to plot on a map or something. (In fact, it is in the Datafiles_2.0/LearningPython directory of the Datafiles_2.0 package you can download). The file is called station.list and its first five lines are:

```
   9.02920   38.76560 2442 AAE
  42.63900   74.49400 1645 AAK
  37.93040   58.11890  678 ABKT
  51.88370 -176.68440  116 ADK
 -13.90930 -171.77730  706 AFI
```

Here is a script (ReadStations.py that will open a file station.list, read in the data and print it out line by line.

```
#!/usr/bin/env python
f=open('station.list')
```

```
StationNFO=f.readlines()
for line  in StationNFO:
    print line
```

If you run this script, you will get this behavior:

```
% ReadStations.py
   9.02920   38.76560 2442 AAE

  42.63900   74.49400 1645 AAK

  37.93040   58.11890  678 ABKT

  51.88370 -176.68440  116 ADK
  etc.
```

The function open() has some bells and whistles to it and has the form open(name[, mode[, buffering]) where the stuff in square brackets is optional. The 'name' argument is the file name to open and 'mode' is the way in which it should be opened, most commonly for reading 'r', writing 'w' or appending 'a'. I use the form 'rU' for unformatted reading because I often want to read in files that were saved in Dos, Mac OR Unix line endings and 'rU' figures all that out for you. Just in case you are curious, Unix lines end in '\n', Mac files in '\r' and Dos (and windows) lines end in '\r\n'. I never use the 'buffering' argument and don't know what it does.

If you are curious about the line endings, try typing out the 'representation' of the line repr(line) in the above script and you get all the stuff that is normally invisible like the apostrophes and the line terminations:

```
% ReadStations.py
'   9.02920   38.76560 2442 AAE \n'
'  42.63900   74.49400 1645 AAK \n'
'  37.93040   58.11890  678 ABKT\n'
'  51.88370 -176.68440  116 ADK \n'
' -13.90930 -171.77730  706 AFI \n'
etc.
```

Notice how in our first version, printing the line also printed the line feed (\n) as an extra line. To clean this off of each line, we can use the string strip() function:

```
    print line.strip('\n')
```

Putting this into the code results in this behavior:

```
% ReadStations.py
   9.02920   38.76560 2442 AAE
  42.63900   74.49400 1645 AAK
  37.93040   58.11890  678 ABKT
```

Let's say you want to read in the data table into lists called Lats, Lons, and StaIDs (the first three columns). You need to split each line into its columns and append the correct column into the appropriate list. Some languages automatically split on the spaces but Python reads in the entire line as a string and ignores the spaces or other possible delimiters (commas, semi-colons, tabs, etc.). To split the line, we use the string function split([sep]) where [sep] is an optional separator. If no separator is specified (e.g., line.split()), it will split on spaces. Anything could be a separator, but the most common ones are ',', ';', and '\t'. The latter is how a tab appears if you were to, say, print out the representation of the line, which shows all the invisibles.

Here is a slightly modified version of ReadStations.py, ParseStations.py which parses out the lines and puts numbers (floats or integers) in the right lists:

```
#!/usr/bin/env python
Lats,Lons,StaIDs,StaName=[],[],[] ,[]# creates lists to put things in
StationNFO=open('station.list').readlines() # combines the open and readlines methods!
for line  in StationNFO:
    nfo=line.strip('\n').split() # strips off the line ending and splits on spaces
    Lats.append(float(nfo[0])) # puts float of 1st column into Lats
    Lons.append(float(nfo[1]))# puts float of 2nd column into Lons
    StaIDs.append(int(nfo[2])) # puts integer of 3rd column into StaIDs
    StaName.append(nfo[3])# puts the ID string into StaName
    print Lats[-1],Lons[-1],StaIDs[-1] # prints out last thing appended
```

From Standard Input

Python can also read from standard input. To do this, we need a system specific module, called sys which among other things has a stdin method. So, instead of specifying a file name in the open command, we could substitute the following line:

```
#!/usr/bin/env python
import sys
Lats,Lons,StaIDs,StaName=[],[],[] ,[]# creates lists to put things in
StationNFO=sys.stdin.readlines() # reads from standard input
for line  in StationNFO:
    nfo=line.strip('\n').split() # strips off the line ending and splits on space
    Lats.append(float(nfo[0])) # puts float of 1st column into Lats
    Lons.append(float(nfo[1]))# puts float of 2nd column into Lons
    StaIDs.append(int(nfo[2])) # puts integer of 3rd column into StaIDs
    StaName.append(nfo[3])# puts the ID string into StaName
    print Lats[-1],Lons[-1],StaIDs[-1] # prints out last thing appended
```

The program can be invoked with:

```
% ReadStations.py < station.list
```

We could also use command line switches by reading in arguments from the command line. In the following example, we use the switch '-f' with the following argument begin the file name:

Command line switches

```
#!/usr/bin/env python
import sys
Lats,Lons,StaIDs,StaName=[],[],[] ,[]# creates lists to put things in
if '-f' in sys.argv:  # look in list of command line arguments
    file=sys.argv[sys.argv.index('-f')+1] # find index of '-f' and increment by
StationNFO=open(file,'rU').readlines() # open file
for line  in StationNFO:
    nfo=line.strip('\n').split() # strips off the line ending and splits on space
    Lats.append(float(nfo[0])) # puts float of 1st column into Lats
    Lons.append(float(nfo[1]))# puts float of 2nd column into Lons
    StaIDs.append(int(nfo[2])) # puts integer of 3rd column into StaIDs
    StaName.append(nfo[3])# puts the ID string into StaName
    print Lats[-1],Lons[-1],StaIDs[-1] # prints out last thing appended
```

This version can be invoked with:

```
% ReadStations.py -f  station.list
```

Reading numeric files

In the special case where the data in a file are entirely numeric, you can read in the file with a special numpy function loadtxt(). This reads the data into a list whereby each element of the list is a list of numbers from each line.

Writing data out

Let's say I have a Python module that will convert latitudes and longitudes to UTM coordinates. O.K. I really do have one that I downloaded from here:

http://code.google.com/p/pyproj/issues/attachmentText?id=27&aid=-8088417477181756 4&name=UTM.py&token=46ab62caa041c3f240ca0e55b7b25ad6

I wrote a script (ConvertStations.py) to convert each of the stations in my list to their UTM equivalents (assuming these were in a WGS-84 ellipsoid). It would be nice if after having done this to the data, I could then write it out somehow, preferably to a file. Of course I could use the print command like this:

```
#!/usr/bin/env python
import UTM # imports the UTM module
Ellipsoid=23-1 # UTMs code for WGS-84
StationNFO=open('station.list').readlines()
for line  in StationNFO:
    nfo=line.strip('\n').split()
    lat=float(nfo[0])
    lon=float(nfo[1])
    StaName= nfo[3]
    Zone,Easting, Northing=UTM.LLtoUTM(Ellipsoid,lon,lat)
    print StaName, ': ', Easting, Northing, Zone
```

which spits out something like this:

```
% ConvertStations.py
AAE :  474238.170087 998088.469113 37P
AAK :  458516.115522 4720850.45385 43T
ABKT :  598330.712671 4198681.92944 40S
ADK :  521722.179764 5748148.625 1U
AFI :  416023.683618 8462168.07766 2L
etc.
```

I could save the output with a UNIX re-direct:

```
ConvertStations.py > mynewfile
```

But we yearn for more. So, more elegantly, I can open an output file [for appending 'a' or (over)writing 'w'] write a formatted string using the write method on the output file object with format string:

```
#!/usr/bin/env python
import UTM # imports the UTM module
outfile=open('mynewfile','w') # creates outfile object
Ellipsoid=23-1 # UTMs code for WGS-84
StationNFO=open('station.list').readlines()
for line  in StationNFO:
    nfo=line.strip('\n').split()
    lat=float(nfo[0])
    lon=float(nfo[1])
    StaName= nfo[3]
    Zone,Easting, Northing=UTM.LLtoUTM(Ellipsoid,lon,lat)
    outfile.write('%s  %s  %s  %s\n'%(StaName, Easting, Northing, Zone))
```

The only significant changes are 1) the object outfile is opened for writing. Note that this will clobber anything in a pre-existing file by that name and 2) the output file gets written to in the statement with a write method on the output file object:

```
    outfile.write('%s  %s  %s  %s\n'%(StaName, Easting, Northing, Zone))
```

The write statement uses the syntax: 'format string'%(list of variables tuple). Format strings have these rules:

- For each variable in (what you...) you need a format: %s for string, %i for integer, %f for float, %e for exponent
- you can also specify further, e.g.: %7.1f for 7 characters with 1 after the decimal %10.3e for 10 characters with 3 after the decimal
- where the number of characters include the decimal and padded spaces
- As noted before, the format string can include punctuation:

```
x,y=4.82,2.3e3
print '%7.1f,%s\t%10.3e'%(x,'hi there',y)
   4.8,hi there  2.300e+03
```

- In the ConvertStations2.py script, the '\n' string puts a UNIX line ending on it. Without that, the whole file is but a single line (very annoying).

A session using the script (ConvertStations2.py and a peek at the resulting file could look like this:

```
% ConvertStations2.py
% head mynewfile
AAE  474238.170087  998088.469113  37P
AAK  458516.115522  4720850.45385  43T
ABKT  598330.712671  4198681.92944  40S
ADK  521722.179764  5748148.625  1U
AFI  416023.683618  8462168.07766  2L
ALE  509467.666259  9161062.29194  20X
ALQ  366981.843985  3868044.56906  13S
ANMO  366981.843985  3868044.56906  13S
ANTO  482347.254856  4413225.7807  36S
AQU  368638.770654  4690300.1797  33T
```

The Unix head command types out the first 10 lines of a file but has no DOS equivalent.

6.7 Functions

So far you have learned how to use functions from program modules like NumPy. You can imagine that there are many bits of code that you might write that you will want to use again and again, say converting between degrees and radians and back, or finding the great circle distance between two points on Earth, or converting between UTM and latitude/longitude coordinates (as in UTM.py, my new favorite package). The basic structure of a program with a Python function is:

```
#!/usr/bin/env python

def FUNCNAME(in_args):
```

```
    """
    DOC STRING
    """
    some code that the functions does something
    return out_args

FUNCNAME(in_args) # this calls the function
```

Line by line analysis

def FUNCNAME(in_args):

The first line must have 'def' as the first three letters, must have a function name with parentheses and a terminal colon. If you want to pass some variables to the function, they go where in_arg sits, separated by commas. There are no output variables here.

There are four different ways to handle argument passing.

1) You could have a function that doesn't need any arguments at all:

```
 #!/usr/bin/env python
def gimmepi():
    """
    returns pi
    """
    return 3.141592653589793
print gimmepi()
```

2) You could use a set list of what are called 'formal' variables that must be passed:

```
#!/usr/bin/env python
def deg2rad(degrees):
    """
    converts degrees to radians
    """
    return degrees*3.141592653589793/180.
print '42 degrees in radians is: ',deg2rad(42.)
```

3) You could have a more flexible need for variables. You signal this by putting *args in the in_args list (along with any formal variables you want):

```
#!/usr/bin/env python
def print_args(*args):
    """
    prints argument list
    """
    print 'You sent me these arguments: '
    for arg in args:
        print arg
print_args(1,4,'hi there')
print_args(42)
```

4) You can use a keyworded, variable-length list by putting **kwargs in for in_args:

```
#!/usr/bin/env python
def print_kwargs(**kwargs):
    """
    prints keyworded argument list
    """
    for key in kwargs:
        print '%s  %s' %(key, kwargs[key])

print_kwargs(arg1='one',arg2=42,arg3='ocelot')
```

Doc String

Although you can certainly write functional code without a document string, make a habit of always including one. Trust me - you'll be glad you did. This can later be used to remind you of what you thought you were doing years later. It can be used to print out a help message by the calling program and it also let's others know what you intended. Notice the use of the triple quotes before and after the documentation string - that means that you can write as many lines as you want.

Function body

This part of the code must be indented, just like in a for loop, or other block of code.

Return statement

You don't need this unless you want to pass back information to the calling body (see, for example print_kwargs() above). Python separates the entrance and the exit. See how it can be done in the gimme_pi() example above.

Main program as function

It is considered good Python style to treat your main program block as a function too. (This helps with using the document string as a help function and building program documentation in general.) In any case, I recommend that you just start doing it that way too. In this case, we have to call the main program with the final (not indented) line main():

```
#!/usr/bin/env python
def print_kwargs(**kwargs):
    """
    prints keyworded argument list
    """
    for key in kwargs:
        print '%s  %s' %(key, kwargs[key])

def main():
    """
    calls function print_kwargs
    """
    print_kwargs(arg1='one',arg2=42,arg3='ocelot')
main()  # runs the main program
```

Notice how in the above examples, all the functions preceded the main function. This is because Python is an interpreter and not compiled - so it won't know about anything declared below as it goes through the script line by line. On the other hand, we've been running lots of functions and they were not in the program we used to call them. The trick here is that you can put a bunch of functions in a separate file (in your path) and import it, just like we did with NumPy. Your functions can then be called from within your program in the same way as for NumPy.

So let's say I put all the above functions in a file called myfuncs.py:

```
def gimmepi():
```

```
    """
    returns pi
    """
    return 3.141592653589793
def deg2rad(degrees):
    """
    converts degrees to radians
    """
    return degrees*3.141592653589793/180.
def print_args(*args):
    """
    prints argument list
    """
    print 'You sent me these arguments: '
    for arg in args:
        print arg
```

I could then just import the module myfuncs from within another program, or just interactively. I can use the functions, or just call for help:

```
% python
>>> import myfuncs
>>> print myfuncs.gimmepi()
3.14159265359
>>> print myfuncs.print_args.__doc__

    prints argument list

>>>
```

Scope of variables

Inside a function, variable names have their own meaning which in many cases will be different from inside the calling function. So, variables names declared inside a function stay in the function. This is true unless you declare them to be "global". Here is an example in which the main program "knows" about the functions variable V:

```
def myfunc():
    global V
```

```
    V=123
def main():
    myfunc()
    print V
main()
```

In addition to being able to write your own functions, of course Python has LOTS of modules and a gazzillion functions. The enthought distribution that you are using Includes plotting, numerical recipes, trig functions, image manipulation, animation, and many more.

6.8 Classes

Before we go any further, we need to learn some basic concepts about classes. These are the basis of "object oriented programming" OOP (that again!). Class objects lie behind plotting, for example and a rudimentary understanding of what they are and how they work will come in handy when we start doing anything but the simplest plotting exercises.

A class object is created by a call to a "class definition" which which can be thought of as a blueprint for the class object. Here is an simple example of a class definition:

```
class Circle:
    """
    This is simple example of a class
    """
    pi=3.141592653589793
    def __init__(self,r):
        self.r=r
    def area(self):
        return 0.5*self.pi*self.r**2
    def circumference(self):
        return 2.*pi*self.r
```

Saving this class in a file called Shapes.py we can use it in a Python session in a manner similar to function modules:

```
>>> import Shapes # import the class
>>> r=4.0
```

```
>>> C=Shapes.Circle(r) # create a class instance with r=4.
>>> C.pi  # retrieve an attribute
3.141592653589793
>>> C.area() # retrieve a method
25.132741228718345
>>> C.r=2.0 # change the value of r
>>> C.area() # get a new area
6.283185307179586
```

In spite of superficial similarities, classes are not the same as functions. Although the Shape module is imported just the same as any other, to use it, we first have to create a class "instance" (C=Shapes.Circle(r)). C is an object with "attributes" (variables) and "methods". All methods (parts that start with "def"), have an argument list. The first argument has to be a reference to the class instance itself, or "self", followed by any variables you want to pass into the method. So the __init__ method initializes the instance attributes of an object. In the above case, it defined the attribute r, which gets passed in when the class is first called. Asking for any attribute (note the lack of parentheses), retrieves the current value of that attribute. Attributes can be changed (as in C.r=2.0).

The other methods (area and circumference) are defined like any function except note the use of 'self' as the first argument. This is required in all class method definitions. In our case, no other parameters are passed in because the only one used is r, so the argument list consists of only self. Calling these methods returns the current values of these methods.

You can make a subclass (child) of the parent class which has all the attributes and methods of the parent, but may have a few attributes and methods of its own. You do this by setting up another class definition within a class.

So, the bottom line about classes is that they are in the same category of things as variables, lists, dictionaries, etc. That is, they are 'data structures' - they hold data, and the methods to process that data. If you are curious about classes, there's lots more to know about classes that we don't have time to get into, but you can find useful tutorials online:

(e.g., `http://www.sthurlow.com/python/lesson08/`)

6.9 Matplotlib

So far you have learned the basics of Python, and NumPy. But Python was sold as a way of visualizing data and we haven't yet seen a single plot (except

a stupid one in the beginning). There are many plotting options within the Python umbrella. The most mature and the one I am most familiar with is matplotlib, a popular graphics module of Python. Actually matplotlib is a collection of a bunch of other modules, toolkits, methods and classes. For a fairly complete and readable tour of matplotlib, check out these links:

`http://matplotlib.sourceforge.net/Matplotlib.pdf`

and here:

`http://matplotlib.sourceforge.net/`

A first plot

Let's start by reviewing the simple plot script (matplotlib1.py):

```
#!/usr/bin/env python
import matplotlib
matplotlib.use("TkAgg") # my favorite backend
import pylab # module with matplotlib
pylab.plot([1,2,3])  # plot some numbers
pylab.ylabel('Y') # label the y-axis
pylab.show() # reveal the plot
```

The first step should be obvious by now, it imports matplotlib. Figures are rendered on "backends" so they appear on screen. There are a lot of different back-ends with slightly different looks. Some work better on different operating systems. I use the very old school backend called "TkAgg" backend because it "works". So step 2 sets the backend: matplotlib.use("TkAgg"). The module matplotlib itself contains a lot of other modules. One of these, pylab is the "business end" that has a lot of plotting methods and classes. It must be loaded alongside matplotlib, so step 3 is: import pylab. After that the fun starts.

In the above example, we call the plot method with a list as an argument. As I mentioned, matplotlib uses the concept of "classes" to make plots and this has just happened behind the scenes. We could have named the plot instance with a the figure() method (e.g., fig=pylab.figure()) and then referred to it later with the command fig.plot([1,2,3]), but we don't have to in this simple case - the class instance is implied and is the "current plot". You can tell this, if you do the above example in interactive mode:

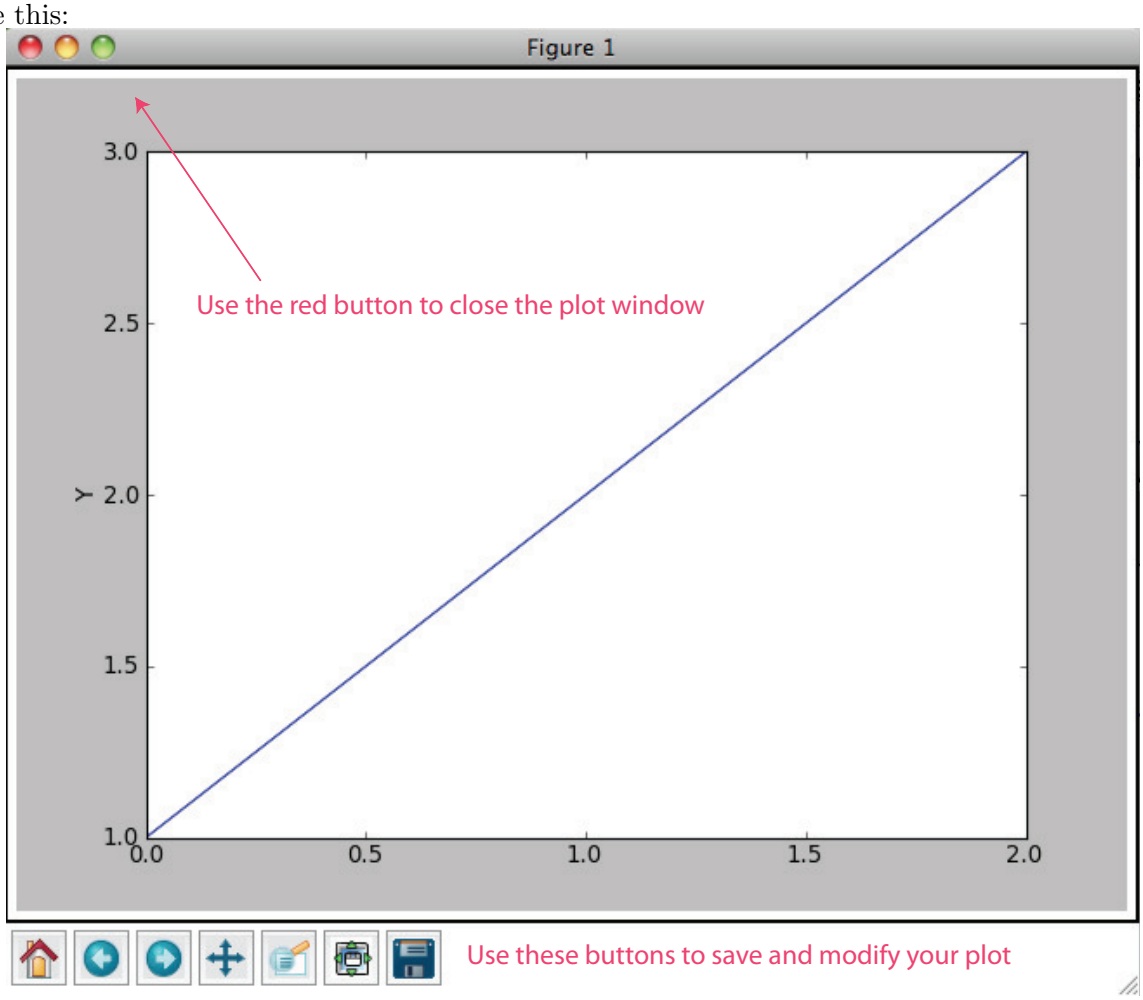
```
>>> import matplotlib
>>> matplotlib.use("TkAgg")
>>> import pylab
```

```
>>> pylab.plot([1,2,3])
[<matplotlib.lines.Line2D object at 0x4bd6eb0>]
```

The bit about [<matplotlib.lines.Line2D object at 0x4bd6eb0>] is Python's way of telling you that you just created an object and something about it. In any case, when you give plot() a single sequence of values (as above), it assumes they are y values and supplies the x values for you.

Attributes of the pylab class, such as the Y axis label can be changes with the ylabel method. As you can imagine, there are LOTS of methods, including, surprise, an xlabel method.

When we are done customizing the plot instance, we can view it with the show method. When that gets executed, we will get a plot something like this:



Once that happens, we won't be able to change the plot any more and in fact, we won't get our terminal back until the little plot window is closed. You can save your plot with the little disk icon in a variety of formats. Adobe Illustrator likes .svg, or .eps while Microsoft products like .png file formats.

If you find it annoying to always have to close figures with the little red button, or save them with the disk icon, you can tweak the program like this:

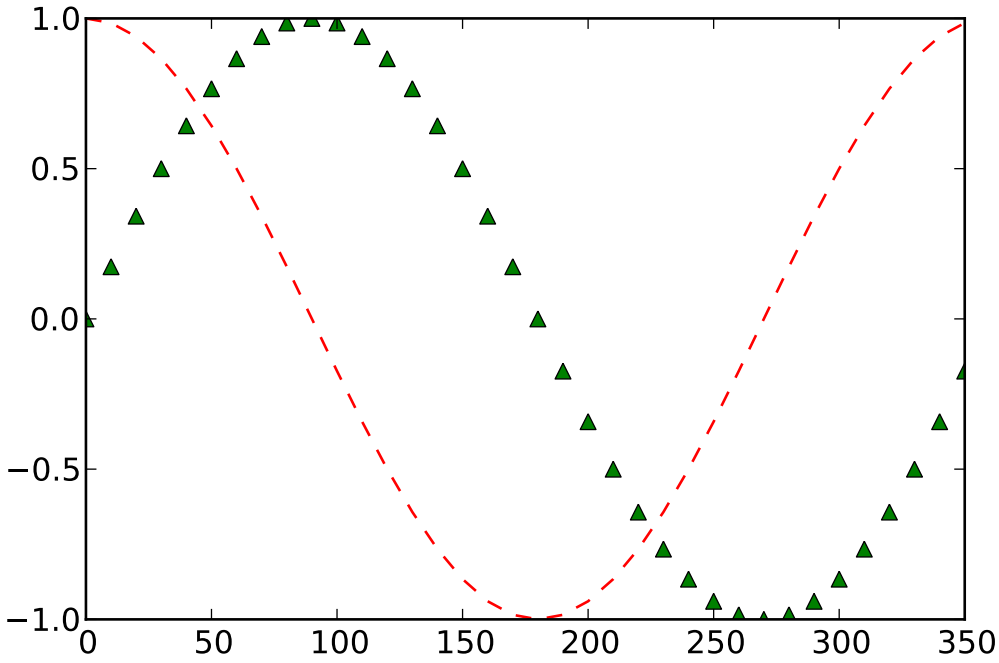
```
#!/usr/bin/env python
import matplotlib
matplotlib.use("TkAgg")
import pylab
pylab.ion()  # turn on interactivity
pylab.plot([1,2,3])
pylab.ylabel('Y')
pylab.draw() # draw  the current plot
ans=raw_input('press [s] to save figure, any other key to quit: ')
if ans=='s':
    pylab.savefig('myfig.eps')
```

The method pylab.savefig(FILENAME.FMT). The .FMT can be one of several, e.g., .eps, .svg, .ps, .pdf, .png, .gif, .jpg, etc.). Some of these (the vector graphics ones like pdf, ps, eps and svg) can be opened in Adobe Illustrator for modification.

As mentioned earlier, if you give plot() a single sequence of values, it assumes they are y values and supplies the x values for you. Garbage in, garbage out. But plot() takes an arbitrary number of arguments of the form: (X_1, Y_1, line_style_1, X_2, Y_2, line_style_2, etc.), where 'line_style' is a string that specifies the line style as illustrated in this script called matplotlib2.py

```
#!/usr/bin/env python
import matplotlib
matplotlib.use("TkAgg")
import pylab,numpy
x=numpy.arange(0,360,10)
r=x*numpy.pi/180.
c=numpy.cos(r)
s=numpy.sin(r)
pylab.plot(x,c,'r--',x,s,'g^')
pylab.show()
```

which produces the plot:



From the code, you can probably figure out that a line style of 'r–' is a red dashed line, and 'g^' are green triangles. There are many other attributes that can be controlled: linewidth, dash style, etc. and I invite you to check out the matplotlib documentation.
By now, you should understand enough about classes, keyword argument passing and other pythonalia to be able to figure things out on your own. But don't panic, I'm going to lead you through a few more examples, which I hope will speed you on your plotting way.

Multiple figures and more customization

As already mentioned, pylab has the concept of "current figure" which subsequent commands refer to. In the preceding examples, we only had one figure, so we didn't have to name it, but for fancier figures with several plots, we can create named figure objects by invoking a figure instance:

```
fig = pylab.figure(num=1,figsize=(5,7)).
```

Notice the syntax whereby figsize is a method with width and height (in inches) specified by a tuple and num is the figure number. Notice that these are keyword arguments, and that there are many more: consult the list of **kwargs in the online documentation located here:

`http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.figure`

Once we have a figure instance (sometimes called a "container"), we can do all kinds of things, including adding subplots. To do this, we can use the syntax:

```
fig.add_subplot(211)
```

Here the argument 211 means 2 rows, one column and this is the first plot. To make plots side by side, you would use: fig.add_subplot(121) for 1 row, two columns, etc.

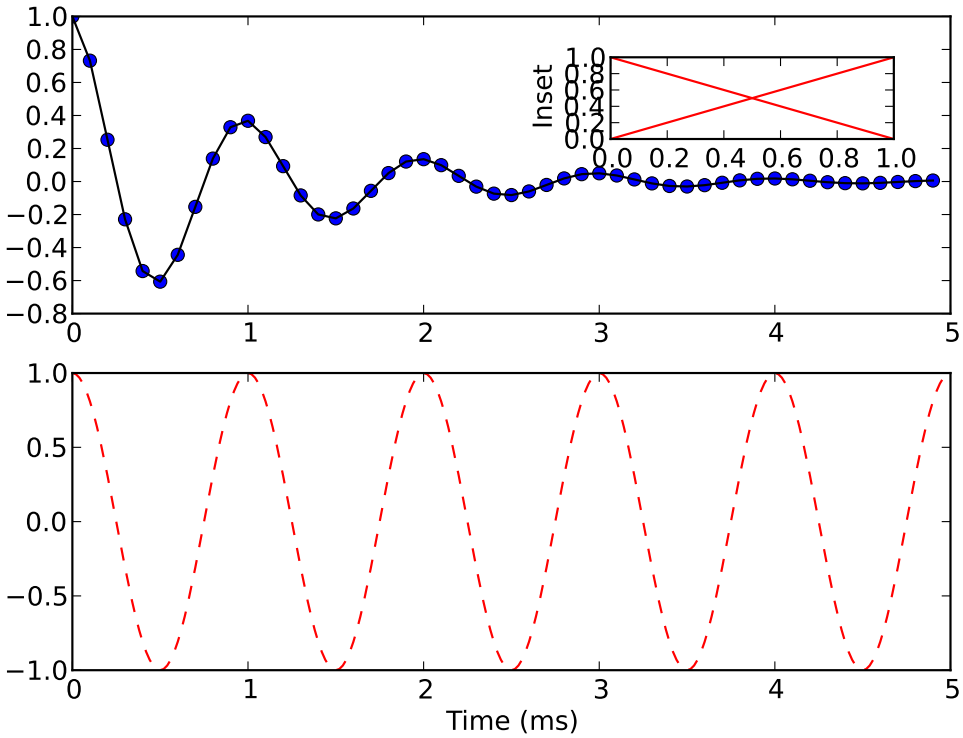
After each add_subplot command, that subplot becomes the current figure for plotting on. If you want more freedom, say, you want to make a subplot at an arbitrary place, use the add_axes([left, bottom,width, height]) 0 method, e.g., add_axes([0.1,0.1,0.7,0.3]). The values are 0-1 in relative figure coordinates.

To illustrate these new concepts, consider the example code, matplotlib3.py:

```
#!/usr/bin/env python
import matplotlib
matplotlib.use("TkAgg")
import pylab, numpy
def f(t):
    return numpy.exp(-t)*numpy.cos(2.*numpy.pi*t)
t1= numpy.arange(0.,5.,0.1)
t2= numpy.arange(0.,5.,0.02)
fig=pylab.figure(num=1,figsize=(7,5))
fig.add_subplot(211)
pylab.plot(t1,f(t1),'bo')
pylab.plot(t1,f(t1),'k-')
fig.add_subplot(212)
pylab.plot(t2,numpy.cos(2*numpy.pi*t2),'r--')
pylab.xlabel('Time (ms)')
fig.add_axes([.6,.75,.25,.10])
pylab.plot([0,1],[0,1],'r-',[0,1],[1,0],'r-')
```

```
pylab.ylabel('Inset')
pylab.show()
```

which produces:



By now, you should be able to figure out what everything in that script does by yourself!

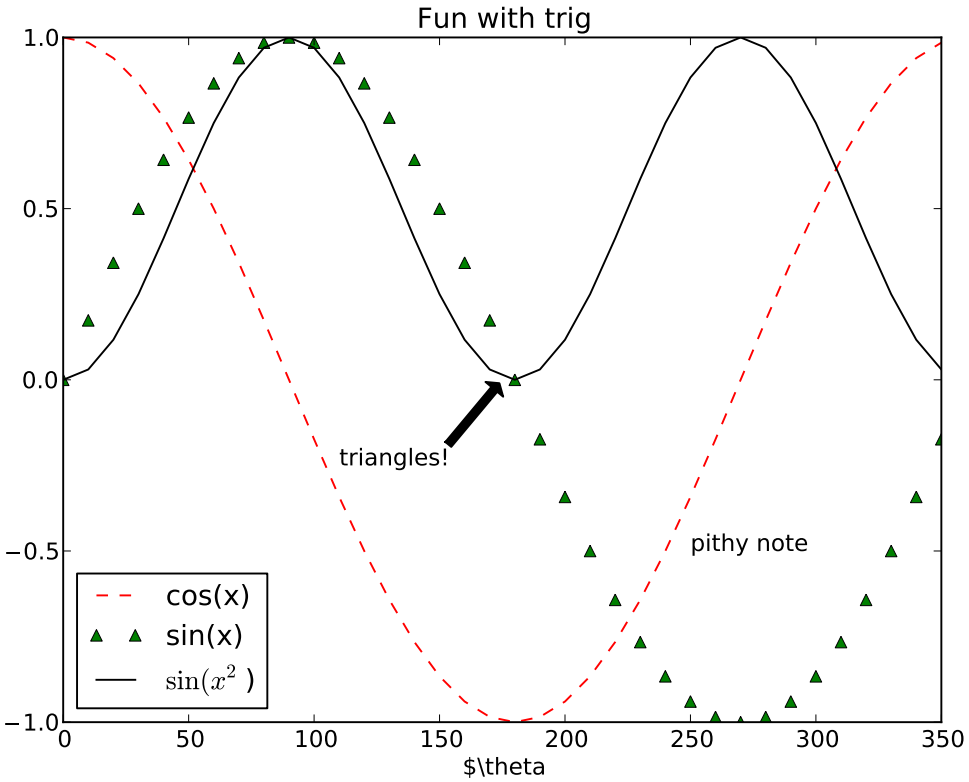
Adding text

We already met xlabel and ylabel. But text can be added in a other ways, e.g., using the title, text, legend and arrow methods. Let's decorate one of our early examples to show how some of these things work:

```
#!/usr/bin/env python
import matplotlib
matplotlib.use("TkAgg")
import pylab,numpy
x=numpy.arange(0,360,10)
```

```
r=x*numpy.pi/180.
c=numpy.cos(r)
s=numpy.sin(r)
s2=numpy.sin(r)**2
pylab.plot(x,c,'r--',x,s,'g^',x,s2,'k-')
pylab.title('Fun with trig')
pylab.text(250,-.5,'pithy note')
pylab.legend(['cos(x)',\
    'sin(x)',r'$\sin(x^2$)'],'lower left')
pylab.xlabel(r'$\theta')
pylab.annotate('triangles!',\
  xy=(175,0),xytext=(110,-.25),\
   arrowprops=dict(facecolor='black',\
   shrink=0.05))
pylab.show()
```

which produces this plot:



The title appears at the top of the plot. Text labels get places at the x and y coordinates on the plot and the legend will appear in the upper/lower right/left corner as specified in the string. The pylab.text(x,y,string, kwargs) method also has optional key word arguments, specifying font, size, color and the like. The legend 'labelist' is a list of labels for each plot element. So, every line or point style that you want in your legend, append a label to the label list after the relevant plot command. Also note that the legend and xlabel methods use a special format for strings (r'LateX String') which allows embedded LaTeX equation syntax to make scientific equations look right - so now you have to learn LaTeX!. Finally, the arrow gets drawn with the annotate method, which has a lot of other attributes as well. Check the matplotlib documentation for details.

There are lots of graphing styles possible with matplotlib, e.g., histograms, pie charts, contour plots, whisker plots, etc. I'm just going to show you a few examples. The best thing to do is to look through the online documentation for a plot that looks like what you need, then modify it. This

is ALWAYS a good approach - start with something that works and fiddle with it until it suits your own particular needs.

Although there is much much more to do in Python, this documentation is aimed at getting and using **PmagPy**, so that's it for this chapter. Congratulations if you made it to the end!